BC Basis Programming Interfaces

HELP.BCDWBLIB

Release 4.6C

Copyright

© Copyright 2001 SAP AG. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP AG. The information contained herein may be changed without prior notice.

Some software products marketed by SAP AG and its distributors contain proprietary software components of other software vendors.

Microsoft®, WINDOWS®, NT®, EXCEL®, Word®, PowerPoint® and SQL Server® are registered trademarks of
Microsoft Corporation.

IBM®, DB2®, OS/2®, DB2/6000®, Parallel Sysplex®, MVS/ESA®, RS/6000®, AIX®, S/390®, AS/400®, OS/390®, and OS/400® are registered trademarks of IBM Corporation.

ORACLE® is a registered trademark of ORACLE Corporation.

INFORMIX®-OnLine for SAP and Informix® Dynamic Server™ are registered trademarks of Informix Software Incorporated.

UNIX®, X/Open®, OSF/1®, and Motif® are registered trademarks of the Open Group.

HTML, DHTML, XML, XHTML are trademarks or registered trademarks of W3C®, World Wide Web Consortium,
Massachusetts Institute of Technology.

JAVA® is a registered trademark of Sun Microsystems, Inc.

JAVASCRIPT® is a registered trademark of Sun Microsystems, Inc., used under license for technology invented and implemented by Netscape.

SAP, SAP Logo, R/2, RIVA, R/3, ABAP, SAP ArchiveLink, SAP Business Workflow, WebFlow, SAP EarlyWatch, BAPI, SAPPHIRE, Management Cockpit, mySAP.com Logo and mySAP.com are trademarks or registered trademarks of SAP AG in Germany and in several other countries all over the world. All other products mentioned are trademarks or registered trademarks of their respective companies.

Icons

Icon	Meaning
	Caution
	Example
	Note
	Recommendation
	Syntax

Contents

BC Basis Programming Interfaces

BC Basis Programming Interfaces

BC Basis Programming Interfaces

BC Basis Programming Interfaces

Changes in Release 4.0A

Background Processing Programming Interface

- External commands, see Sample Program: Adding a Job Step for an External Command or Program [Page 31]
- Automatic mailing of job output to a SAPoffice user, see Sample Program: Immediate Start with Spool Request Mail Recipient [Page 35]

Programming with the Background Processing System

This section describes the programming interface of the background processing system. You can use this interface to schedule and manage background processing jobs from your applications.

Programming Techniques

With the programming interface of the background processing system, you can schedule and manage background jobs from your own programs.

You wish to make a log-running report available to users from a menu entry. In the transaction that underlies the menu entry, you could do the following:

- optionally collect job specifications, such as the start date and time, from the user, and
- schedule the job for execution by the background processing system.

You can use the programming interface to start both ABAP programs and programs that are external to the R/3 System.

The programming interface also offers function modules for the following tasks:

- managing jobs (displaying, copying, and deleting jobs)
- checking and triggering events, which can be used to start other background jobs
- displaying the log generated by a background job.

The following sections explain the most important tasks in scheduling and managing background jobs.

Scheduling and Managing a Job: Easy Method

The programming interface offers two function modules for “easy” scheduling and management of background jobs. These function modules offer less options -- less programmer control -- over scheduling and management, but are easy to use. They’re intended for use if you simply want to schedule a job with a minimum of fuss and programming effort.

The function modules are as follows:

- BP_JOBVARIANT_SCHEDULE [Page 13]: Schedule a job for execution.

  This function module greatly simplifies scheduling a job. You need only name an ABAP report. The function module presents screens to your user to allow them to 1) specify the variant to use; and 2) pick a single or repetitive start time for the job.

- BP_JOBVARIANT_OVERVIEW [Page 15]: Manage jobs.

  This function module offers a simplified management interface to jobs. A user can delete inactive jobs and display job logs and spool output from completed jobs.

Easy Job Scheduling Using BP_JOBVARIANT_SCHEDULE

To schedule a job from within a program using the express method, you need only call the BP_JOBVARIANT_SCHEDULE function module.

The express method has the following characteristics:

- Simplified job structure: The function module schedules a job that includes only a single job step.
- The function module uses default values for most job-processing options. You cannot, for example, specify a target printer as part of the call to the function module. Instead, the job step uses the print defaults of the scheduling user.
- Only ABAP reports can be scheduled. You must use the “full-control” method to start external programs.
- The range of start-time options is restricted. Event-based scheduling is not supported.

The function module works as follows:

1. You name the report that is to be scheduled in your call to the function module.
2. The function module displays a list of variants to the user. The user must select a variant for the report.

   You must ensure that the variants required by your users have already been defined.
3. The user picks either “immediate start” or enters a start date and start time. Optionally, the user can also make the job restart periodically. The job is then scheduled.

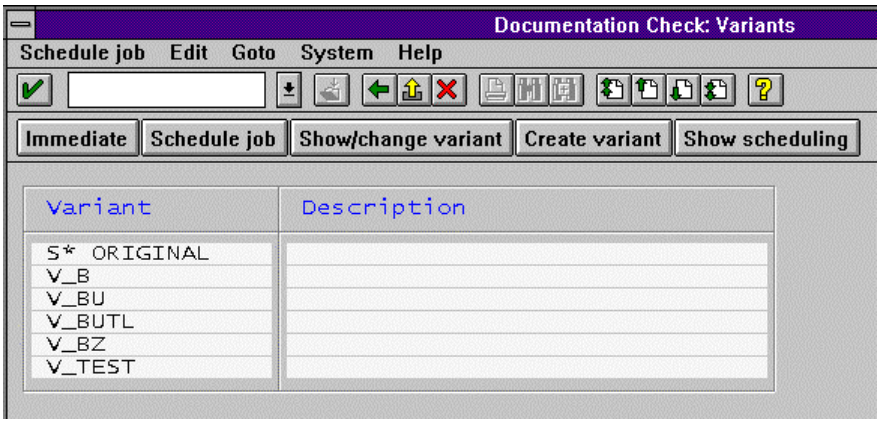
Example

You could use the following code to let users schedule report RSTWGZS2 for checking on the status of online documentation:

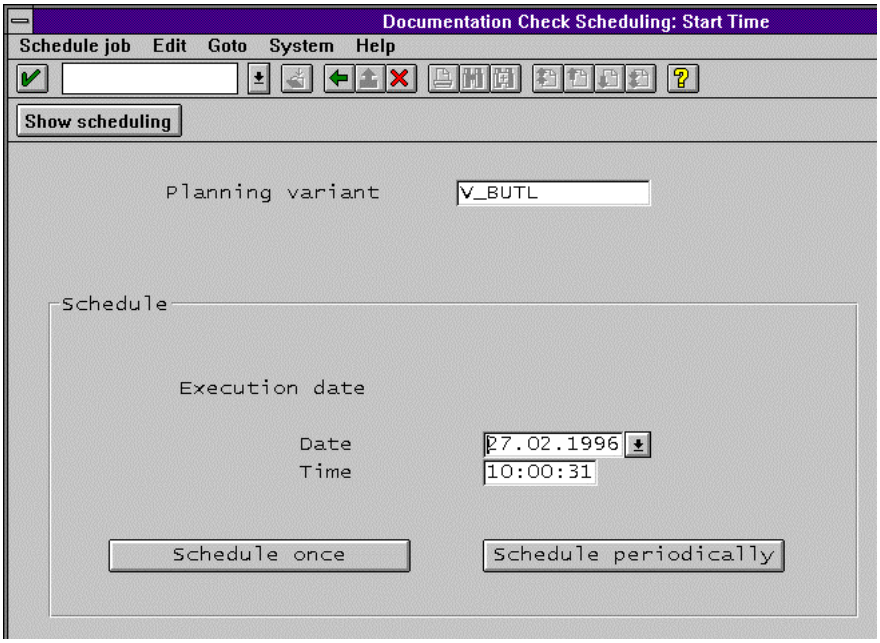
```
call function 'BP_JOBVARIANT_SCHEDULE'
   exporting
     title_name  = 'Documentation Check'   " Displayed as title of
                                           " of scheduling screens
     job_name    = 'DocuCheck'             " Name of background
                                           " processing job
     prog_name   = 'RSTWGZS2'              " Name of ABAP
                                           " report that is to be
                                           " run -- used also to
                                           " select variants
   exceptions
     no_such_report = 01.                  " PROG_NAME program
                                           " not found.
call function 'BP_JOBVARIANT_OVERVIEW'     " List the jobs that
   exporting                               " have been scheduled
     title_name  = 'Documentation Check'   " Displayed as title
                                           " of overview screen
     job_name    = 'DokuCheck'             " Jobs with this name
                                           " are listed
```

```
    prog_name = 'RSTWGZS2'
  exceptions
    no_such_job = 01.
```

The code would present your users with a sequence of two screens. On the first screen, shown below, the user can select from available variants or add new variants:



On the second screen, the user can pick a start date and time for the job and decide whether the job should be repeated periodically. The function module finishes with a success message when the user schedules the background job.



See also Managing “Easy-Method” Jobs with BP_JOBVARIANT_OVERVIEW [Page 15] for an easy method for letting users manage their background jobs.

Managing “Easy-Method” Jobs with BP_JOBVARIANT_OVERVIEW

With BP_JOBVARIANT_OVERVIEW, you can offer your users a simplified display and management function for their “express-method” jobs.

The function module does the following:

- displays the job title, report name and variant, start time specification, status, and scheduling user of each selected “express” job
- allows users to change the start specification and restart period of jobs that have not yet been run
- allows a user to display the job log and spool output of completed jobs
- allows users to delete jobs that are not currently executing.

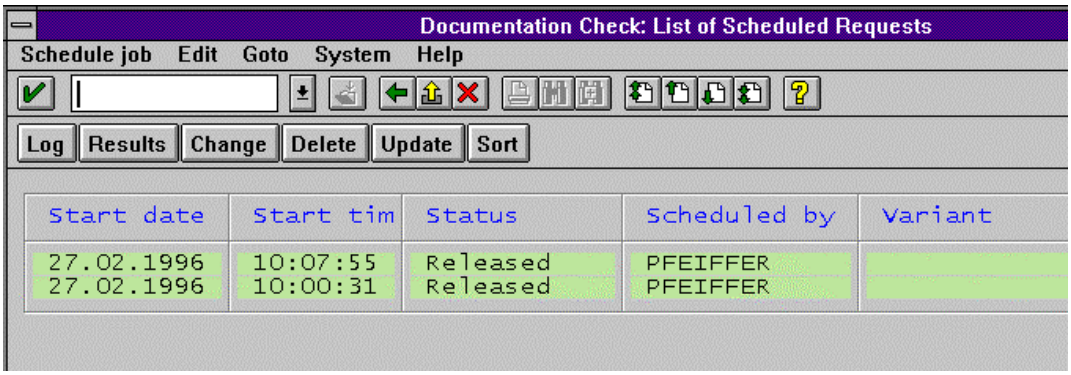
Example

You could use the following code to let users check on jobs that they have scheduled with the sample code shown in Easy Job Scheduling Using BP_JOBVARIANT_SCHEDULE [Page 13]:

```
call function 'BP_JOBVARIANT_SCHEDULE'     " Schedule jobs;
                                           " counterpart to
   exporting                               " BP_JOBVARIANT_OVERVIEW
     title_name  = 'Documentation Check'   " Displayed as title of
                                           " of scheduling screens
     job_name    = 'DocuCheck'             " Name of background
                                           " processing job
     prog_name   = 'RSTWGZS2'              " Name of ABAP
                                           " report that is to be
                                           " run -- used also to
                                           " select variants
   exceptions
     no_such_report = 01.                  " PROG_NAME program
                                           " not found.

call function 'BP_JOBVARIANT_OVERVIEW'
   exporting
     title_name  = 'Documentation Check'   " Displayed as title
                                           " of overview screen
     job_name    = 'DokuCheck'             " Jobs with this name
                                           " are listed
     prog_name = 'RSTWGZS2'
   exceptions
     no_such_job = 01.
```

The user is presented with the following screen:

Managing “Easy-Method” Jobs with BP_JOBVARIANT_OVERVIEW



From this screen, your users can do the following, subject to background processing authorizations:

- delete or change the start time of jobs that have not yet started
- display the job logs and spool output generated by jobs that have been completed.

Scheduling a Job: Full-Control Method

Scheduling a background job instructs the background processing system to process the programs included in the job. One or more ABAP and/or external programs can be included in a job.

There are two ways to schedule a job from a program:

- the “express” method using the BP_JOBVARIANT_SCHEDULE function module

  For more information, please see Scheduling and Managing a Job: Easy Method [Page 12].

- the “full-control” method using the JOB_OPEN, JOB_SUBMIT, and JOB_CLOSE function modules.

  With this method, you have full control over such job options as printing, start time options, and so on.

The following sections describe the full-control method for scheduling a job.

Task Overview: Full-Control Job Scheduling

To schedule a background job from within a program using the full-control method, you must do the following:

1. Create the job using function module JOB_OPEN.

   JOB_OPEN passes a name to the background system. The system returns the job count, the unique ID number that identifies the job, together with the job name.

2. Collect the job specifications (report or program to run, start time, and so on).

   You can collect the job specifications on your own. Or you can use the following function modules to collect the specifications:

   – BP_START_DATE_EDITOR: Ask the user to specify a start time for the background job.

   – GET_PRINT_PARAMETERS: Ask the user for printing and optical archiving specifications. You can also use GET_PRINT_PARAMETERS to make these specifications yourself, in non-interactive mode.

     GET_PRINT_PARAMETERS is not essential in test programs or in programs that do not produce any printable output (no WRITE keywords). For example, a background job that is to make only a set of database changes need not include a call to GET_PRINT_PARAMETERS.

     However, you should always include a call to GET_PRINT_PARAMETERS in any production program for scheduling jobs that produce output. Otherwise, the R/3 spool system uses defaults for formatting and printing any output generated by a job. The printouts that result may not be what you wish.

3. Add a job step to the job with the function module JOB_SUBMIT or the ABAP keyword SUBMIT.

   Each of the programs that is included in a job forms a separate job step. For each job step, you can make separate specifications for attributes other than the start date and target system for running a job. For example, each job step in a job can have separate printing and authorizations specifications.

   You can add as many job steps to a job as you wish. You can mix job steps that start ABAP and external programs.

   There is only one difference between JOB_SUBMIT and ABAP SUBMIT. With SUBMIT, you can specify selection criteria for an ABAP report dynamically, when the background job is submitted. With JOB_SUBMIT, you can only specify an existing variant for a report.

   You must use SUBMIT, for example, if the user of your program supplies the selection criteria for an ABAP report interactively, at runtime. With JOB_SUBMIT, you could specify only an existing variant for the report.

4. Close the job and pass it to the background processing system for execution with the function module JOB_CLOSE. You specify job-start attributes and many run-time attributes of a job in JOB_CLOSE.

   A job cannot be started until you have "closed" it.

With JOB_CLOSE, you can find out whether a job was only scheduled or was also released to run. If a job was only scheduled, then an administrator must release the job before it can run. If a job was automatically released, then the background processing system can start the job as soon as its specified start time is reached.

A job is not released but is only scheduled if either of the following is true:

- The scheduling user does not have release authorization.
- The job does not have a start-time specification.

If you are scheduling a job that is to be periodically repeated, you can use the BP_CALCULATE_NEXT_JOB_STARTS function module to find the dates upon which the job would be scheduled to run.

Where to Specify Job Attributes

The following table shows for each attribute of a job:

- whether the attribute applies to a job as a whole or only to a single job step

  Each of the programs in a job forms a separate job step. For each job step, you can make separate specifications for attributes other than the start date and target system for running the job.

- where and how the value for each attribute is assigned.

For programming examples of the attributes, please see the next section. Values for most of the attributes are predefined in the ABAP module LBTCHDEF.

Where to Specify Job Attributes

Attribute	Applies to Job or to Job Step	Where to Specify
Job name	Job	JOB_OPEN, parameter JOBNAME [Page 29]
Job number	Job	JOB_OPEN (returned by system), parameter JOBCOUNT [Page 29]
Job priority	Job	Always set by default to class C. No other priority can be set with the background programming interface. You can set other priorities only by hand, in the job-scheduling transaction (SM36, also SM37).
Immediate start	Job	JOB_CLOSE, parameter STRTIMMED [Page 34]
Start and repeat specifications	Job	JOB_CLOSE, various parameters. See the programming samples in Task Overview: Full-Control Job Scheduling [Page 18]
Target system for execution	Job	JOB_CLOSE, parameter TARGETSYSTEM [Page 37]
Automatic deletion after job has run	Job	JOB_OPEN, parameter DELANFREP
Recipient for mailed spool requests generated by job	Job	JOB_CLOSE, parameter RECIPIENT_OBJ [Page 35]
Name of report to run	Job step	JOB_SUBMIT [Page 30], ABAP SUBMIT [Page 33]
Variant for report	Job step	JOB_SUBMIT, parameter VARIANT, ABAP SUBMIT, parameter USING SELECTION-SET
Values for selection variables	Job step	ABAP SUBMIT, various parameters. See The ABAP User's Guide [Ext.]

Name and operating-specific variant of external command to run	Job step	JOB_SUBMIT, COMMANDNAME and OPERATINGSYSTEM parameters [Page 31]
Name of external program to run	Job step	JOB_SUBMIT, EXTPGM_NAME [Page 31] parameter
Argument string for external program	Job step	JOB_SUBMIT, EXTPGM_PARAM [Page 31] parameter
Target system for external program	Job step	JOB_SUBMIT, EXTPGM_SYSTEM [Page 31] parameter
Control options for external program	Job step	JOB_SUBMIT, various parameters. See Sample Program: Adding a Job Step for an External Command or Program [Page 31]
User for job run-time authorizations	Job step	JOB_SUBMIT, AUTHCKNAM [Page 30] parameter. ABAP SUBMIT
Printing and archiving specifications	Job step	JOB_SUBMIT, ABAP SUBMIT using GET_PRINT_PARAMETERS [Page 27] to supply specifications

Getting Job-Start Specifications from Users

You can use the BP_START_DATE_EDITOR function module to have your users specify when and how a job is to be started. The function module offers the same range of scheduling choices that a user has with the standard scheduling transaction: immediate start, start window, predecessor job, event, and so on.

Sample Program

This section contains examples for BP_START_DATE_EDITOR and JOB_CLOSE.

```
* Data structure returned by BP_START_DATE_EDITOR
*
DATA STARTSPECS LIKE TBTCSTRT.
DATA START_MODIFY_FLAG LIKE BTCH0000-INT4.
*
* BP_START_DATE_EDITOR:  Present user with pop-up window
* requesting specifications for the job start.  The user can
* select from
* - immediate start,
* - start time and date and last time and date,
* - start after event,
* - start after activation of a new operating mode,
* - start after predecessor job
* - start on a certain workday of the month, counted from the
*   start or end of the month
* - specify how a job is to be handled if the start date falls
*   on a holiday.
*
* All start date options are selectable by the user.  To limit the
* selection available to the user, you should program your own
* input window.  Or, you can evaluate TBTCSTRT-STRTDTTYP to see
* if the user chose an appropriate start specification (see
* below).
*
* BP_START_DATE_EDITOR checks for plausibility and other errors in
* user specifications and issues an error if any problems exist.

* Use only the TBTCSTRT fields shown in the example below.  Other
* fields are reserved for internal use only.  Do not set TBTCSTRT
* fields directly.
*
CALL FUNCTION 'BP_START_DATE_EDITOR'
   EXPORTING
     STDT_DIALOG      = BTC_YES      " Module in interactive mode
     STDT_OPCODE      = BTC_EDIT_STARTDATE " Edit mode
     STDT_INPUT       = STARTSPECS    " Table for user selections
     STDT_TITLE       = 'Title'      " Title for pop-up screen
   IMPORTING
     STDT_OUTPUT      = STARTSPECS    " User selections
     STDT_MODIFY_TYPE = START_MODIFY_FLAG
                                      " Flag: did user change start
*                                     " specifications?  Values:
```

```
                                        " - BTC_STDT_MODIFIED,
                                        "   user did change specs
                                        " - BTC_STDT_NOT_MODIFIED,
                                        "   user did not change specs
   EXCEPTIONS
     OTHERS           = 99.
*
* Flag for specifying immediate start in JOB_CLOSE:  For the
* immediate-start case only, you must set a flag for communicating
* the user selection to JOB_CLOSE.  In all other cases, simply
* pass the values returned by BP_START_DATE_EDITOR to JOB_CLOSE.
* Those that were not set by the user have the value SPACE and
* have no effect in JOB_CLOSE.
*
* Setting all JOB_CLOSE parameters is only permissible when you
* use BP_START_DATE_EDITOR.  Otherwise, you should set only the
* required parameters in your call to JOB_CLOSE.
*
DATA: STARTIMMEDIATE LIKE BTCH0000-CHAR1.

CASE STARTSPECS-STARTDTTYP. " Possible types are listed below.
  WHEN BTC_STDT_IMMEDIATE. " User selected immediate start.
    STARTIMMEDIATE = 'X'.
  WHEN BTC_STDT_DATETIME.  " User entered start date and time
  WHEN BTC_STDT_EVENT.     " User entered event and possibly
                           " argument OR user selected start on
                           " activation of a particular operation
                           " mode (job start event driven in this
                           " case as well).
    <Optional error processing, if you wish to prevent user from
    scheduling a job dependent upon an event>
  WHEN BTC_STDT_AFTERJOB.   " User entered predecessor job.
    <Optional error processing, if you wish to prevent user from
    scheduling a job dependent upon a predecessor job>
  WHEN BTC_STDT_ONWORKDAY   " User selected a job start on a
                            " particular workday of the month.
ENDCASE.
*
* Use the start specifications provided by the user in JOB_CLOSE.
*
  CALL FUNCTION 'JOB_CLOSE'
   EXPORTING
     JOBNAME     = JOBNAME
     JOBCOUNT    = JOBNUMBER
     STRTIMMED   = STARTIMMEDIATE
     SDLSTRTDT   = STARTSPECS-SDLSTRTDT
     SDLSTRTTM   = STARTSPECS-SDLSTRTTM
     LASTSTRTDT  = STARTSPECS-LASTSTRTDT
     LASTSTRTTM  = STARTSPECS-LASTSTRTTM
     PRDDAYS     = STARTSPECS-PRDDAYS    " Non-zero values in
     PRDHOURS    = STARTSPECS-PRDHOURS   " any PRD* field mean
     PRDMINS     = STARTSPECS-PRDMINS    " that a startdate,
```

Getting Job-Start Specifications from Users

```
      PRDMONTHS    = STARTSPECS-PRDMONTHS  " starttime job is to

      PRDWEEKS     = STARTSPECS-PRDWEEKS   " be repeated
                                           " periodically.
      TARGETSYSTEM = STARTSPECS-INSTNAME
*     AT_OPMODE    = Omit this parameter if you obtain user
                   " specifications.  It's set automatically by
                   " BP_START_DATE_EDITOR.
      AT_OPMODE_PERIODIC = STARTSPECS-PERIODIC  " Set with generic
                                                " periodic flag in
                                                " table TBTCSTRT.
      PRED_JOBNAME = STARTSPECS-PREDJOB
      PRED_JOBCOUNT = STARTSPECS-PREDJOBCNT
      PREDJOB_CHECKSTAT = STARTSPECS-CHECKSTAT
      EVENT_ID     = STARTSPECS-EVENTID
      EVENT_PARAM = STARTSPECS-EVENTPARM
      EVENT_PERIODIC = STARTSPECS-PERIODIC  " Set with generic
                                            " periodic flag in
                                            " table TBTCSTRT.
      CALENDAR_ID = STARTSPECS-CALENDARID
      STARTDATE_RESTRICTION = STARTSPECS-PRDBEHAV
      START_ON_WORKDAY_NOT_BEFORE = STARTSPECS-NOTBEFORE
      START_ON_WORKDAY_NR = STARTSPECS-WDAYNO
      WORKDAY_COUNT_DIRECTION = STARTSPECS-WDAYCDIR
                   " START_ON_WORKDAY jobs are scheduled
                   " automatically for periodic execution if
                   " PRDMONTHS is set.
    IMPORTING
      JOB_WAS_RELEASED = JOB_RELEASED
    EXCEPTIONS
      OTHERS       = 99.
```

Calculating Start Dates for Periodic Jobs

You can use BP_CALCULATE_NEXT_JOB_STARTS to find the dates upon which a particular periodic job would be started. The function module takes the identification of a single job, from JOB_OPEN. Start dates and times are returned in an internal table of format TBTCJOB.

Obtaining Printing and Archiving Specifications

If an ABAP program in a background job produces list output, then this output is deposited in a spool request in the R/3 spool system.

You can specify how this spool request is to be processed with the GET_PRINT_PARAMETERS. You can either specify your own printing and archiving specifications in non-interactive mode or display the standard printing/archiving pop-up to your users.

For a sample program, please see Sample Program: Declarations, Set Up, and Print Specifications [Page 27].

You cannot directly modify the data in the printing and archiving structures that GET_PRINT_PARAMETERS maintains. You must use GET_PRINT_PARAMETERS to fill these structures.

More Information on Printing/Archiving Specifications

You should always include a call to GET_PRINT_PARAMETERS in any job-scheduling program that is to be used in production systems.

The only exceptions are for the following:

- programs that schedule jobs that do not produce output (no WRITE keyword). For example, a program that schedules a job that in turn makes only a series of database changes does not need a call to GET_PRINT_PARAMETERS.
- test programs. A test program also can run without a call to GET_PRINT_PARAMETERS.

It is important to use GET_PRINT_PARAMETERS because otherwise the spool system uses default values for preparing output. The spool system reads any default values for target printer and spool request disposition from the user master record of the submitting user and formats the output for printing at 80 characters per line and 59 lines per page. There is no guarantee that the resulting formatting will correspond to the printing requirements of the output. For example, the default line length may not correspond to the line length required by the output.

When you call GET_PRINT_PARAMETERS, you'll need to specify that the function module is to run in BATCH mode and you'll need to provide the name of the report that is to be run in the particular job step. The function module searches the report for LINE-SIZE and LINE-COUNT specifications in the REPORT keyword. These values, if present, are used as defaults for printing. They are also used, together with the target printer type, to select the most appropriate format for the output. If the destination printer is specified interactively by your user, then the function module presents a pop-up to confirm the format selection.

Sample Program: Declarations, Set Up, and Print Specifications

```
REPORT BCSAMPLE.
*
* You must include LBTCHDEF in all background processing
* applications.  Use only the LBTCHDEF definitions that appear
* in examples as values for function modules.  LBTCHDEF also
* declares values that are used only internally in background
* processing.
*
* If you plan to schedule external programs, you must include
* RSXPGDEF to define constants for the control flags for external
* programs.
*
INCLUDE LBTCHDEF.      " Background processing definitions.
INCLUDE RSXPGDEF.      " Definitions for external programs.
*
* Sample DATA entries for scheduling a job.
*
  DATA: JOBNUMBER       LIKE TBTCJOB-JOBCOUNT,  " Job ID and
        JOBNAME         LIKE TBTCJOB-JOBNAME,   " job name.
        STARTDATE       LIKE TBTCJOB-SDLSTRTDT, " Start-time
        STARTTIME       LIKE TBTCJOB-SDLSTRTTM, " window specs.
        LASTSTARTDATE   LIKE TBTCJOB-LASTSTRTDT,
        LASTSTARTTIME   LIKE TBTCJOB-LASTSTRTTM,
        JOB_RELEASED    LIKE BTCH0000-CHAR1.   " JOB_CLOSE: Was
                                               " job released?
*
* The data structures used by background jobs are:
* TBTCJOB:    Job definition
* TBTCSTRT:   Job start time (with function module
*             BP_START_DATE_EDITOR)
*
* In any production job-scheduling program, you should call
* GET_PRINT_PARAMETERS to specify printing and archiving
* parameters.  See Getting Job-Start Specifications from Users [Page 22].
*
* The following printing and archiving tables are
* used by background jobs.
* PRI_PARAMS: Printing options
* ARC_PARAMS: Archiving options
*
* Structure for print parameters
*
  DATA USER_PRINT_PARAMS LIKE PRI_PARAMS.
*
* Structure for optical archiving parameters
*
  DATA USER_ARC_PARAMS LIKE ARC_PARAMS.
*
```

Sample Program: Declarations, Set Up, and Print Specifications

```
* Additional printing/archiving declarations
*
  DATA:         COUNT(3) TYPE N VALUE 1,
                VALID    TYPE C.

* GET_PRINT_PARAMETERS:  As coded, presents the user with the
* standard R/3 popup window for collecting printer and optical
* archiving specifications.  You can export default values for
* printing and archiving options.  These values are presented as
* defaults in interactive mode or can be written directly into
* the parameter structures in non-interactive mode.  Use the
* IMPORTING OUT* parameters to pass the printing and archiving
* specifications to JOB_SUBMIT or ABAP SUBMIT.

  CALL FUNCTION 'GET_PRINT_PARAMETERS'
     EXPORTING
       MODE     = 'BATCH'
       REPORT   = '<REPORT NAME>'
       NO_DIALOG = ' '
     IMPORTING
       OUT_PARAMETERS         = USER_PRINT_PARAMS
       OUT_ARCHIVE_PARAMETERS = USER_ARC_PARAMS
       VALID                  = VALID
     EXCEPTIONS
       OTHERS = 99.

  IF VALID = SPACE.
     <Terminate program, user aborted print option selection>
  ENDIF.
```

Sample Program: Creating a Job with JOB_OPEN

```
*
* Create your job with JOB_OPEN.  The module returns a unique job
* number.  Together with the jobname, this number identifies the
* job.  Other parameters are available, but are not required.
*
  JOBNAME = 'Freely selectable name for the job(s) you create'.

  CALL FUNCTION 'JOB_OPEN'
   EXPORTING
     JOBNAME          = JOBNAME
   IMPORTING
     JOBCOUNT         = JOBNUMBER
   EXCEPTIONS
     CANT_CREATE_JOB  = 01
     INVALID_JOB_DATA = 02
     JOBNAME_MISSING  = 03
     OTHERS           = 99.

  IF SY-SUBRC > 0.
    <Error processing>
  ENDIF.
```

Sample Program: Adding an ABAP Job Step

```
*
* Add a job step: ABAP program
*
  CALL FUNCTION 'JOB_SUBMIT'
   EXPORTING
     AUTHCKNAM       = SY-UNAME           " Runtime authorizations
                                          " user
     JOBCOUNT        = JOBNUMBER          " Value from JOB_OPEN
     JOBNAME         = JOBNAME            " Value from JOB_OPEN
     REPORT          = 'REPORT'           " Report to be run
     VARIANT         = 'VARIANT'          " Variant to use with
                                          " report
     PRIPARAMS       = USER_PRINT_PARAMS  " User printing options
     ARCPARAMS       = USER_ARC_PARAMS    " User archiving options
                                          " Both sets of options
                                          " come from
                                          " GET_PRINT_PARAMETERS
   EXCEPTIONS
     BAD_PRIPARAMS            = 01
     INVALID_JOBDATA          = 02
     JOBNAME_MISSING          = 03
     JOB_NOTEX                = 04
     JOB_SUBMIT_FAILED        = 05
     LOCK_FAILED              = 06
     PROGRAM_MISSING          = 07
     PROG_ABAP_AND_EXTPG_SET  = 08
     OTHERS                   = 99.

  IF SY-SUBRC > 0.
    <Error processing>
  ENDIF.
* Error processing in the event of a non-recoverable error should
* include a call to BP_JOB_DELETE with FORCEDMODE = 'X' to remove
* the job from the database.  This is required for error
* processing in all calls after JOB_OPEN.  (The background system
* cleans up automatically in the event of a problem in JOB_OPEN).
*
* Please see Selecting and Deleting a Job [Page 53] for a programming
* example.
*
```

Sample Program: Adding a Job Step for an External Command or Program

With Release 4.0, there are now two ways to schedule external programs or commands in a job step. These are:

- External commands -- pre-defined symbolic names for commands or programs at the operating system level. External commands are intended for ordinary users. Because they are pre-defined in R/3 and are authorizations-tested, the administrator has control over what users can do at the operating system level through background jobs.
- External programs -- OS commands that are directly entered into the job step specifications and for which no specific authorization test is carried out in R/3. Use of external programs is restricted to administrators (a special authorization is required.

You can schedule either an external command or an external program in a single job step, but not both. To schedule an external command, you use the COMMANDNAME and OPERATING SYSTEM parameters. To schedule an external program, you use the EXTPGM_NAME parameter.

Both external commands and external programs share the EXTPGM-parameters other than EXTPGM_NAME.

For more information, search for “External Programs” in the CCMS Guide [Ext.].

```
*
* Add a job step: external program
*
  CALL FUNCTION 'JOB_SUBMIT'
   EXPORTING
     AUTHCKNAM       = SY-UNAME              " User for runtime
                                             " authorizations
     JOBCOUNT        = JOBNUMBER             " From JOB_OPEN
     JOBNAME         = JOBNAME               " From JOB_OPEN
     COMMANDNAME     = EXTERNAL_COMMAND      " Name of a pre-defined
                                             " external command.
                                             " COMMANDNAME and
                                             " EXTPGM_NAME are
                                             " mutually-exclusive
                                             " alternatives.  Both
                                             " use the EXTPGM
                                             " parameters.
     OPERATINGSYSTEM= 'AIX'                  " Operating system for
                                             " choosing COMMANDNAME
                                             " variant.
     EXTPGM_NAME     = '/usr/exe/myexe'      " Pathname of an
                                             " external program;
                                             " an authorization
                                             " for S_RZL_ADM (CC
                                             " Control Center:
                                             " System Administration)
                                             " is required.
     EXTPGM_PARAM    = '<Parameter String>'  " Program parameters
     EXTPGM_SYSTEM      = 'host01'           " Host for execution
```

```
    EXTPGM_WAIT_FOR_TERMINATION = 'X'      " Control flags for
    EXTPGM_STDOUT_IN_JOBLOG = 'X'          " external programs:
    EXTPGM_SET_TRACE_ON     = 'X'          " see RSXPGDEF
    EXTPGM_STDERR_IN_JOBLOG = 'X'          " documentation for
                                           " EXTPGM options
  EXCEPTIONS
    INVALID_JOBDATA         = 02
    JOBNAME_MISSING         = 03
    JOB_NOTEX               = 04
    JOB_SUBMIT_FAILED       = 05
    LOCK_FAILED             = 06
    PROGRAM_MISSING         = 07
    PROG_ABAP_AND_EXTPG_SET = 08
    BAD_XPGFLAGS            = 09
    OTHERS                  = 99.

IF SY-SUBRC > 0.
  <Error processing>
ENDIF.
```

Sample Program: Adding a Job Step with ABAP SUBMIT

```
*
* Add a job step:  ABAP SUBMIT keyword
* SUBMIT is a fully-compatible alternative to the JOB_SUBMIT
* function module.  SUBMIT allows specification of a variant
* or direct specification of values for parameters. See the ABAP
* syntax documentation (help submit in the ABAP editor) for more
* information.
*
  SUBMIT REPORTNAME AND RETURN
     USER SY-UNAME           " User for runtime authorizations
     VIA JOB JOBNAME NUMBER JOBNUMBER
                             " Job name and job number
                             " from JOB_OPEN
     TO SAP-SPOOL            " Print and archiving options from
                             " GET_PRINT_PARAMETERS
                             " Both sets of options come from
                             " GET_PRINT_PARAMETERS
        SPOOL PARAMETERS USER_PRINT_PARAMS
        ARCHIVE PARAMETERS USER_ARC_PARAMS
        WITHOUT SPOOL DYNPRO.
```

Sample Program: Immediate Start with JOB_CLOSE

```
*
* Submit job for processing:  immediate start
*
  CALL FUNCTION 'JOB_CLOSE'
   EXPORTING
     JOBCOUNT          = JOBNUMBER     " Job identification: number
     JOBNAME           = JOBNAME       " and name.
     STRTIMMED         = 'X'           " Schedules the job for
                                       " immediate start.  The job
                                       " is started immediately
                                       " only if the user has the
                                       " RELE authorization to
                                       " release a job to run.
   IMPORTING
     JOB_WAS_RELEASED = JOB_RELEASED " If user has authorization
                                     " to release jobs to run, job
                                     " is automatically released
                                     " when it is scheduled.  This
                                     " field is set to 'x' if the
                                     " job has been released.
                                     " Otherwise, the job is sche-
                                     " duled but must be released
                                     " by an administrator before
                                     " it can be started.
   EXCEPTIONS
*    CANT_START_IMMEDIATE  No longer used.  Replaced by IMPORTING
*                          parameter JOB_WAS_RELEASED.
     INVALID_STARTDATE    = 01
     JOBNAME_MISSING      = 02
     JOB_CLOSE_FAILED     = 03
     JOB_NOSTEPS          = 04
     JOB_NOTEX            = 05
     LOCK_FAILED          = 06
     OTHERS               = 99.

  IF SY-SUBRC > 0.
    <Error processing>
  ENDIF.
```

Sample Program: Immediate Start with Spool Request Mail Recipient

With Release 4.0, you can automatically mail spool requests that are generated by a background job to a SAPoffice user.

The following sample program shows how to generate the SWOTOBJID structure that JOB_CLOSE needs to identify the SAPoffice user. For more information, please see the ABAP documentation on programming with Business Object Repository objects.

```
INCLUDE LBTCHDEF.
*
* Include the Business Object Repository object
INCLUDE <CNTN01>.

DATA BEGIN OF LOCAL_JOB.
        INCLUDE STRUCTURE TBTCJOB.
DATA END OF LOCAL_JOB.

DATA BEGIN OF LOCAL_STEP_TBL OCCURS 10.
        INCLUDE STRUCTURE TBTCSTEP.
DATA END OF LOCAL_STEP_TBL.

*
* Data declarations for the mail recipient
DATA RECIPIENT  TYPE SWC_OBJECT.
DATA RECIPIENT_OBJ LIKE SWOTOBJID.
SWC_CONTAINER CONTAINER.

* Data declarations for the background job
EXECHOST LIKE SY-HOST DEFAULT 'host1',
EXESERVR LIKE BTCTGTSRVR-SRVNAME DEFAULT 'host1_SID_00',
ABAPNAME LIKE SY-REPID DEFAULT 'Program2run',
EMPFNAME LIKE SY-UNAME DEFAULT 'JSmith'.

* Generate recipient object (see report RSSOKIF1 for an example)
*** 1. Create the recipient:

IF EMPFNAME <> SPACE.
   "** 1.1 Generate an object reference to a recipient object
   SWC_CREATE_OBJECT RECIPIENT 'RECIPIENT' SPACE.

   "** 1.2 Write the import parameters for method
   "** recipient.createaddress into the container
   "** and empty the container
   SWC_CLEAR_CONTAINER CONTAINER.
   "** Set address element (internal user 'JSMITH')
   SWC_SET_ELEMENT CONTAINER 'AddressString' EMPFNAME.
   "** Set address type (internal user)
   SWC_SET_ELEMENT CONTAINER 'TypeId' 'B'.

   "** 1.3 Call the method recipient.createaddress
```

Sample Program: Immediate Start with Spool Request Mail Recipient

```
   SWC_CALL_METHOD RECIPIENT 'CreateAddress' CONTAINER.
   "** Issue any error message generated by a method exception
   IF SY-SUBRC NE 0.
      MESSAGE ID SY-MSGID TYPE 'E' NUMBER SY-MSGNO.
   ENDIF.
   SWC_CALL_METHOD RECIPIENT 'Save' CONTAINER.
   SWC_OBJECT_TO_PERSISTENT RECIPIENT RECIPIENT_OBJ.
ENDIF.
*** Recipient has been generated and is ready for use in a
*** background job

CALL FUNCTION 'JOB_OPEN'
...

CALL FUNCTION 'JOB_SUBMIT'
...

IF EMPFNAME <> SPACE.
   CALL FUNCTION 'JOB_CLOSE'
     EXPORTING
          JOBCOUNT                    = LOCAL_JCOUNT
          JOBNAME                     = LOCAL_JOBNAME
          STRTIMMED                   = 'X'
          TARGETSYSTEM                = EXECHOST
          TARGETSERVER                = EXESERVR
          RECIPIENT_OBJ               = RECIPIENT_OBJ
     IMPORTING
          JOB_WAS_RELEASED            = JOBLOS
     EXCEPTIONS
          CANT_START_IMMEDIATE        = 1
          INVALID_STARTDATE           = 2
          JOBNAME_MISSING             = 3
          JOB_CLOSE_FAILED            = 4
          JOB_NOSTEPS                 = 5
          JOB_NOTEX                   = 6
          LOCK_FAILED                 = 7
          OTHERS                      = 8.

   IF ( SY-SUBRC <> 0).
      WRITE: / 'Job_Close(empfname) Problem:', SY-SUBRC.
   ELSE.
      WRITE: / 'Job_Close(empfname) done'.
   ENDIF.

ELSE. "no empfname
....
ENDIF.
```

Sample Program: Start-Time Window with JOB_CLOSE

```
*
* Submit job:  start-time window defined, with periodic repeat and
* with target system upon which the job is to run.
*
* In this case, you must provide start date settings.  You can
* have the job started immediately with a JOB_CLOSE parameter,
* set the start date yourself, or ask the user with
* BP_START_DATE_EDITOR.
*
* Example:  Provide start-date specifications yourself.
*
  STARTDATE = '19970101'.
  STARTTIME = '160000'.
  LASTSTARTDATE = '19970101'.
  LASTSTARTTIME = '180000'.

  CALL FUNCTION 'JOB_CLOSE'
   EXPORTING
     JOBCOUNT     = JOBNUMBER     " Job identification: number
     JOBNAME      = JOBNAME       " and name.
     SDLSTRTDT    = STARTDATE     " Start of start-time...
     SDLSTRTTM    = STARTTIME     " window
     LASTSTRTDT   = LASTSTARTDATE " Optional: end of...
     LASTSTRTTM   = LASTSTARTTIME " start-time window
     PRDMONTHS    = '1'           " Restart at intervals of
     PRDWEEKS     = '1'           " the sum of the PRD*
     PRDDAYS      = '1'           " parameters
     PRDHOURS     = '1'
     PRDMINS      = '1'
     STARTDATE_RESTRICTION = BTC_DONT_PROCESS_ON_HOLIDAY
                             " Restrict job start to work
                             " days; don't start job if
                             " scheduled on holiday.
                             " Other values:
                             " BTC_PROCESS_BEFORE_HOLIDAY
                             " BTC_PROCESS_AFTER_HOLIDAY
     CALENDAR_ID  = '01'     " ID of R/3 factory calendar
                             " for determining workdays
     TARGETSYSTEM = 'hs0011' " Optional: name of the host
                             " system on which the job is
                             " to run.  Set only if
                             " absolutely required.
                             " Example: A required tape
                             " drive is accessible only
                             " from a single host system.
   IMPORTING
     JOB_WAS_RELEASED = JOB_RELEASED " If user has authorization
                                     " to release jobs to run, job
                                     " is automatically released
                                     " when it is scheduled.  This
```

```
                                    " field is set to 'x' if the
                                    " job has been released.
                                    " Otherwise, the job is sche-
                                    " duled but must be released
                                    " by an administrator before
                                    " it can be started.
 EXCEPTIONS
   INVALID_STARTDATE    = 01
   JOBNAME_MISSING      = 02
   JOB_CLOSE_FAILED     = 03
   JOB_NOSTEPS          = 04
   JOB_NOTEX            = 05
   LOCK_FAILED          = 06
   OTHERS               = 99.

IF SY-SUBRC > 0.
  <Error processing>
ENDIF.
```

Sample Program: Job Start on Workday (JOB_CLOSE)

```
* Start on particular workday of each month according to
* factory calendar.
*
* In this example, we hardwire the number of the workday and the
* workday.  These specifications could also be collected from the
* user with BP_START_DATE_EDITOR.
*
  CALL FUNCTION 'JOB_CLOSE'
   EXPORTING
     CALENDAR_ID     = '01' " ID of R/3 factory calendar
                            " for determining workdays
     JOBCOUNT        = JOBNUMBER
     JOBNAME         = JOBNAME
     PRDMONTHS       = 1  " Specify whether job is to be repeated
                          " monthly, bi-monthly, etc.  You may not
                          " use other PRD* fields.  They are
                          " ignored.
     START_ON_WORKDAY_NOT_BEFORE = SY-DATUM
                          " Start job not before the specified
                          " date (today's date)
     START_ON_WORKDAY_NR = '03'
                          " Start job on the third workday in the
                          " factory calendar, with respect to
                          " start or end of month, as specified in
                          " workday_count_direction.
     WORKDAY_COUNT_DIRECTION     = BTC_BEGINNING_OF_MONTH
                          " Specify whether workday number is
                          " relative to start or end of the month.
                          " Permissible values:
                          " - BTC_BEGINNING_OF_MONTH: Third
                          "   workday after the start of the
                          "   month.
                          " - BTC_END_OF_MONTH: Third workday
                          "   before the end of the month.
   IMPORTING
     JOB_WAS_RELEASED = JOB_RELEASED  " Check whether job was
                                      " released.
   EXCEPTIONS
     INVALID_STARTDATE           = 2
     JOBNAME_MISSING             = 3
     JOB_CLOSE_FAILED            = 4
     JOB_NOSTEPS                 = 5
     JOB_NOTEX                   = 6
     LOCK_FAILED                 = 7
     OTHERS                      = 99.
```

Sample Program: Job Start on Event (JOB_CLOSE)

```
*
* Submit job for processing:  job eligible for start when the
* specified event occurs.
*
  CALL FUNCTION 'JOB_CLOSE'
   EXPORTING
     JOBCOUNT        = JOBNUMBER     " Job identification: number
     JOBNAME         = JOBNAME       " and name.
     EVENT_ID        = 'EVENT_NAME'  " Name of the event for which
                                     " the job is to wait.
     EVENT_PARAM       = 'Additional qualifying value'
     EVENT_PERIODIC    = 'X'         " Restart job whenever event
                                     " is triggered.
   EXCEPTIONS
     CANT_START_IMMEDIATE = 01
     INVALID_STARTDATE    = 02
     JOBNAME_MISSING      = 03
     JOB_CLOSE_FAILED     = 04
     JOB_NOSTEPS          = 05
     JOB_NOTEX            = 06
     LOCK_FAILED          = 07
     OTHERS               = 99.

  IF SY-SUBRC > 0.
    <Error processing>
  ENDIF.
```

Sample Program: Wait for Predecessor Job with JOB_CLOSE

```
* Start at successful completion of predecessor job.  In this
* example, the user selects the predecessor job from a list of
* jobs.  The jobs are selected with BP_JOB_SELECT and displayed
* for selection to the user with BP_JOBLIST_PROCESSOR.
*
* Important:  If you specify a predecessor job directly in your
* program (not interactively chosen), then be sure that the
* predecessor job has the status Scheduled or Released when
* you schedule the successor job.  Otherwise, the successor
* scheduling will fail.  That is, a job that is active or
* already completed cannot be specified as a predecessor job.
*
* Data declarations:  BP_JOB_SELECT and BP_JOBLIST_PROCESSOR
*
DATA JSELECT LIKE BTCSELECT.
DATA SEL_JOBLIST LIKE TBTCJOB OCCURS 100 WITH HEADER LINE.
DATA SELECTED_JOB LIKE TBTCJOB.

* Generate list of jobs as with the background processing
* management transaction (SM37).

* Sample selection criteria
*
JSELECT-JOBNAME = 'NAME OF JOB'.
JSELECT-JOBCOUNT = '<Job count>'.
JSELECT-USERNAME = SY-UNAME.

  CALL FUNCTION 'BP_JOB_SELECT'
   EXPORTING
     JOBSELECT_DIALOG    = BTC_YES  " Display list to user
     JOBSEL_PARAM_IN     = JSELECT  " default job selection
                                    " criteria
   IMPORTING
     JOBSEL_PARAM_OUT    = JSELECT  " User's job selection
                                    " criteria
   TABLES
     JOBSELECT_JOBLIST   = SEL_JOBLIST
   EXCEPTIONS
     INVALID_DIALOG_TYPE = 1
     JOBNAME_MISSING     = 2
     NO_JOBS_FOUND       = 3
     SELECTION_CANCELED  = 4
     USERNAME_MISSING    = 5
     OTHERS              = 99.
*
* Ask the user to pick a job from the list.  The selected job is
* the predecessor job.
*
  call function 'BP_JOBLIST_PROCESSOR'
```

Sample Program: Wait for Predecessor Job with JOB_CLOSE

```
   exporting
     joblist_opcode      = btc_joblist_select
                           " Starts processor in mode which
                           " allows user to pick a job from
                           " list.  Selected job is returned
                           " in the joblist_sel_job parameter.
     joblist_refr_param  = jselect
                           " User's job selection parameters
                           " for Refresh function
   importing
     joblist_sel_job     = selected_job
                           " Returns the job selected from the
                           " job list by the user.
   tables
     joblist             = sel_joblist
                           " Input from BP_JOBLIST_SELECT
   exceptions
     invalid_opcode             = 1
     joblist_is_empty           = 2
     joblist_processor_canceled = 3
     others                     = 4.
*
* Schedule job using the predecessor picked by the user.
*
  CALL FUNCTION 'JOB_CLOSE'
   EXPORTING
     JOBCOUNT             = SELECTED_JOB-JOBCOUNT
                            " Job ID from BP_JOBLIST_PROCESSOR
     JOBNAME              = SELECTED_JOB-JOBNAME
                            " Job ID from BP_JOBLIST_PROCESSOR
     PREDJOB_CHECKSTAT    = 'X'         " Start job only if
                                        " predecessor job was
                                        " successfully completed.
     PRED_JOBCOUNT        = SEL_JOBLIST-JOBCOUNT
                                        " ID of predecessor job,
                                        " from BP_JOB_SELECT.
     PRED_JOBNAME         = SEL_JOBLIST-JOBNAME
   IMPORTING
     JOB_WAS_RELEASED     = JOB_RELEASED
                                        " Check: was job released?
   EXCEPTIONS
     CANT_START_IMMEDIATE = 1
     INVALID_STARTDATE    = 2
     JOBNAME_MISSING      = 3
     JOB_CLOSE_FAILED     = 4
     JOB_NOSTEPS          = 5
     JOB_NOTEX            = 6
     LOCK_FAILED          = 7
     OTHERS               = 99.
```

Sample Program: Start at Switch to Operating Mode (JOB_CLOSE)

```
* Start job when a particular R/3 operation mode becomes active.
* Operation modes are defined in the Computing Center Management
* System (CCMS, transaction SRZL).
*
* Note that internally opmode jobs are handled as "start on event"
* jobs.  This means, for example, that after BP_START_DATE_EDITOR,
* TBTCSTRT-STDTTYP is set to BTC_STDT_EVENT when a user selects an
* opmode start.
*
  CALL FUNCTION 'JOB_CLOSE'
   EXPORTING
     JOBCOUNT             = JOBNUMBER
     JOBNAME              = JOBNAME
     AT_OPMODE            = 'NIGHTS' " Job is eligible to
                                     " start when this
                                     " operation mode becomes
                                     " active.
     AT_OPMODE_PERIODIC   = 'X'      " Set this field to X to
                                     " start job each time
                                     " that the op mode
                                     " becomes active.
   IMPORTING
     JOB_WAS_RELEASED     = JOB_RELEASED   " Check: was job
                                           " released?
   EXCEPTIONS
     INVALID_STARTDATE    = 1
     JOBNAME_MISSING      = 2
     JOB_CLOSE_FAILED     = 3
     JOB_NOSTEPS          = 4
     JOB_NOTEX            = 5
     LOCK_FAILED          = 6
     OTHERS               = 99.
```

Job Outcomes: Displaying Job Logs

For each background job that is run, the background processing system generates a log which may contain the following:

- Job-progress messages from the background processing system
- Error messages from the program(s) that were run in the job
- In the case of external programs, any output from the external program that was redirected to the job log.

The following sections explain how to do the following from an ABAP program:

- Allow a user to display a job log
- Read the contents of a job log into an internal table for further processing (creation of a single log for all job steps in a job, for example).

Displaying a Job Log [Page 45]

Copying a Job Log into an Internal Table [Page 46]

Displaying a Job Log

Use function module BP_JOBLOG_SHOW to display a job log to a user.

BP_JOBLOG_SHOW cannot usually be part of the same program that you used to schedule a job, since the job log is complete only when the job has been completed.

However, you could offer display of a job log as a separate function to your users.

You will need the job name and ID number to identify the job log. You must either:

- save these specifications yourself when you schedule a job
- or use BP_JOB_SELECT to find and select from jobs.

Sample Program: Displaying a Job Log

```
REPORT BCSWPTS2.
INCLUDE LBTCHDEF.

* Possible data declarations:  BP_JOBLOG_SHOW
* Assumption:  You have saved the JOBNAME and JOBCOUNT of a job
* and are specifying these values explicitly.

DATA: JOBCOUNT LIKE TBTCJOB-JOBCOUNT.
DATA: JOBNAME LIKE TBTCJOB-JOBNAME.

JOBNAME = '<NAME OF JOB>'.         " Supplied by you when you
                                   " schedule a job.
JOBCOUNT = '<NUMBER OF JOB>'.      " Returned by JOB_OPEN.

CALL FUNCTION 'BP_JOBLOG_SHOW'
   EXPORTING
     CLIENT = SY-MANDT         " Defaults to user's client.
     JOBCOUNT = JOBCOUNT       " Job ID number.
     JOBNAME = JOBNAME         " Job name.
   EXCEPTIONS
     JOBLOG_DOES_NOT_EXIST   = 01  " Log already deleted
     JOBLOG_IS_EMPTY         = 02  " Job has just started. If
                                   " exception recurs, there is
                                   " probably a system problem.
     NO_JOBLOG_THERE_YET     = 03  " Job not yet started.
     NO_SHOW_PRIVILEGE_GIVEN = 04  " Calling user does not have
                                   " display privileges for the
                                   " requested job.
   OTHERS                  = 99.  " System errors, such as
                                   " database or network problems.
```

Copying a Job Log into an Internal Table

Use BP_JOBLOG_READ to read the contents of a job log into an internal table. You can then further process the job log. For example, you could concatenate the logs from several related jobs or search for the term canceled (German abgebrochen) to detect jobs that did not run successfully.

Sample Program

```
REPORT BPJOBLOG.
INCLUDE LBTCHDEF.
*
* Possible data declarations:  BP_JOBLOG_READ
* Assumption:  You have saved the JOBNAME and JOBCOUNT of a job
* and are specifying these values explicitly.
*
* Job log records are returned in an internal table.
*
DATA: JOBNUMBER LIKE TBTCJOB-JOBCOUNT.
DATA: JOBLOGID LIKE TBTCJOB-JOBLOG.
DATA: JOBNAME LIKE TBTCJOB-JOBNAME.

DATA JOBLOG OCCURS 100 LIKE TBTC5.

JOBNAME = '<NAME OF JOB>'.     " Supplied by you when you schedule
                               " a job.
JOBCOUNT = '<NUMBER OF JOB>'. " Returned by JOB_OPEN.

  CALL FUNCTION 'BP_JOBLOG_READ'
   EXPORTING
     CLIENT =    SY-MANDT      " Defaults to user's client.
     JOBCOUNT = JOBNUMBER      " Job ID number.
     JOBNAME =  JOBNAME        " Job name.
   TABLES
     JOBLOGTBL = JOBLOG
   EXCEPTIONS
     JOBLOG_DOES_NOT_EXIST    = 01  " Log already deleted
     JOBLOG_IS_EMPTY          = 02  " Job has just started. If
                                    " exception recurs, there is
                                    " probably a system problem.
     NO_JOBLOG_THERE_YET      = 03  " Job not yet started.
     NO_SHOW_PRIVILEGE_GIVEN = 04   " Calling user does not have
                                    " display privileges for the
                                    " requested job.
     OTHERS                   = 99. " System errors, such as
                                    " database or network
                                    " problems.
```

Managing Jobs: Generating Job Lists

With function module BP_JOB_MAINTENANCE (transaction SM37), you can call the full job maintenance system of the background processing system, starting with the job selection screen.

Since many users are not familiar with job maintenance and have no desire to search for their jobs, you can use the function modules BP_JOB_SELECT and BP_JOBLIST_PROCESSOR to select and display a list of jobs for the users of your program.

Use BP_JOB_SELECT to generate an internal table of jobs. Then, with BP_JOBLIST_PROCESSOR, you can display the selected jobs in the list format used by the job maintenance system.

You can also use BP_FIND_JOBS_WITH_PROGRAM to select jobs that run a particular program. Use this function module with BP_JOBLIST_PROCESSOR to display a job list to your users. Like BP_JOBLIST_SELECT; BP_FIND_JOBS_WITH_PROGRAM offers interactive and silent modes.

For an example that uses BP_FIND_JOBS_WITH_PROGRAM, please see Displaying Job Status: SHOW_JOBSTATE [Page 50].

You can display a job list in three modes:

- Display mode: The user can view job attributes and such job products as spool requests and job logs. He or she cannot, however, change jobs or perform such operations as releasing a job to run.
- Edit mode: The user can perform all maintenance operations for which he or she is authorized.
- Select mode: The user is requested to select a job from the list. When the selection is made, the list is closed and the TBTCJOB structure of the selected job is returned to your program.

Authorizations

The authorization tests for background processing are in effect for processing of job lists. These authorizations are described in the *Users, Authorizations, and System Security* guide and in Basis Customizing.

Sample Program: Generating a Job List [Page 48]

Sample Program: Generating a Job List

```
*
* Data declarations:  BP_JOB_SELECT
*
DATA JSELECT LIKE BTCSELECT.

DATA SEL_JOBLIST LIKE TBTCJOB OCCURS 100.


*
* Data declarations:  BP_JOBLIST_PROCESSOR
*
DATA SELECTED_JOB LIKE TBTCJOB.
*
* Sample selection criteria
*
JSELECT-JOBNAME = 'Generic Name*'.
JSELECT-USERNAME = SY-UNAME.

CALL FUNCTION 'BP_JOB_SELECT'
   EXPORTING
     JOBSELECT_DIALOG            = BTC_NO
     JOBSEL_PARAM_IN             = JSELECT
   IMPORTING
     JOBSEL_PARAM_OUT            = JSELECT
   TABLES
     JOBSELECT_JOBLIST  = SEL_JOBLIST
   EXCEPTIONS NO_JOBS_FOUND      =  1
     SELECTION_CANCELED          =  2
     OTHERS                      = 99.

  IF SY-SUBRC > 0.
    <Error processing>
  ENDIF.

CALL FUNCTION 'BP_JOBLIST_PROCESSOR'
   EXPORTING
     JOBLIST_OPCODE      = BTC_JOBLIST_EDIT " Set mode.
                                            " Other constants:
                                            " BTC_JOBLIST_SHOW
                                            " BTC_JOBLIST_SELECT
     JOBLIST_REFR_PARAM  = JSELECT
   TABLES
     JOBLIST             = SEL_JOBLIST
   IMPORTING                                " Useful only for
     JOBLIST_SEL_JOB     = SELECTED_JOB     " select mode.
   EXCEPTIONS
     INVALID_OPCODE      = 1
     JOBLIST_IS_EMPTY    = 2
     OTHERS              = 99.

  IF SY-SUBRC > 0.
```

```
  <Error processing>
ENDIF.
```

Displaying Job Status: SHOW_JOBSTATE

You can find out the status of a job with the SHOW_JOBSTATE function module. You provide this function module with a job name and job number. It returns one of the six possible statuses of the job.

The following example searches for jobs that contain a particular report or external program, using BP_FIND_JOBS_WITH_PROGRAM. It then uses SHOW_JOBSTATE to check the status of each of the jobs that were found. Using an IF keyword, the program evaluates the statuses returned by the function module and lists them, using icons to represent the statuses.

```
REPORT EXAMPLE.
INCLUDE LBTCHDEF.  " Background processing definitions.
INCLUDE <LIST>.    " List icons, lines, and other definitions.

* Data declaration:  BP_FIND_JOBS_WITH_PROGRAM
DATA SEL_JOBLIST LIKE TBTCJOB OCCURS 100 WITH HEADER LINE.
                   " Internal table for jobs found.

* Data declaration: SHOW_JOBSTATE
DATA:              " Possible job statuses.
    ABORTED TYPE C,
    FINISHED TYPE C,
    PRELIMINARY TYPE C,
    READY TYPE C,
    RUNNING TYPE C,
    SCHEDULED TYPE C.

* Find jobs that contain the ABAP program RSCRIDX4.  The
* program and variant name must be in capital letters.  It's
* also possible to search for jobs that run an external program.
CALL FUNCTION 'BP_FIND_JOBS_WITH_PROGRAM'
     EXPORTING
          ABAP_PROGRAM_NAME              = 'RSCRIDX4'
          ABAP_VARIANT_NAME              = 'V_ALL'
          EXTERNAL_PROGRAM_NAME          = ' '

                   " In non-interactive mode, you can search with
                   " a program name, a variant name, or both.
                   " Omitting all these specifications results in
                   " PROGRAM_SPECIFICATION_MISSING.
          DIALOG                         = BTC_NO
                   " BTC_NO:  non-interactive mode.  You must
                   " specify at least a program name in the call.
                   " BTC_YES:  your user enters the program name
                   " and variant.
     TABLES
          JOBLIST                        = SEL_JOBLIST
                   " Internal table of format TBTCJOB containing
                   " any jobs found.
     EXCEPTIONS
          NO_JOBS_FOUND                  = 1
          PROGRAM_SPECIFICATION_MISSING = 2
```

```
          INVALID_DIALOG_TYPE           = 3
          JOB_FIND_CANCELED             = 4
          OTHERS                        = 5.

IF SY-SUBRC <> 0.
  <Error handling>
ENDIF.

* Write list heading lines.
WRITE: / 'Status',  9 'Job Name',  25 'Job Number'.
ULINE AT /1(85).

* Loop through jobs found by BP_FIND_JOBS_WITH_PROGRAM.
LOOP AT SEL_JOBLIST.

  CALL FUNCTION 'SHOW_JOBSTATE'
       EXPORTING
            JOBCOUNT         = SEL_JOBLIST-JOBCOUNT
            JOBNAME          = SEL_JOBLIST-JOBNAME
                  " Identify the job to be checked.  Both
                  " name and count are required.
       IMPORTING
                  " Possible statuses.  The status of the
                  " job is set to the value 'X'.
            ABORTED          = ABORTED
                  " Job terminated abnormally.
            FINISHED         = FINISHED
                  " Job completed successfully.
            PRELIMINARY      = PRELIMINARY
                  " Job can't be started:  Job scheduled
                  " but not yet released to run or job scheduled
                  " with no start condition.
            READY            = READY
                  " Job scheduled, released, start condition
                  " fulfilled, but job not yet started.
            RUNNING          = RUNNING
                  " Job in progress.
            SCHEDULED        = SCHEDULED
                  " Job scheduled and released, waiting for start
                  " condition to be fulfilled.
       EXCEPTIONS
            JOBCOUNT_MISSING = 1
            JOBNAME_MISSING  = 2
            JOB_NOTEX        = 3
            OTHERS           = 4.

IF SY-SUBRC <> 0.
  <Error handling>
ENDIF.

* Determine the status of the job just checked by
* SHOW_JOBSTATE.  Assign appropriate status icon and
```

Displaying Job Status: SHOW_JOBSTATE

```
* write to a list.

  IF ABORTED = 'X'.
    WRITE: /3 ICON_FAILURE AS ICON,
              AT 9 SEL_JOBLIST-JOBNAME,
              AT 25 SEL_JOBLIST-JOBCOUNT.
  ELSEIF FINISHED = 'X'.
    WRITE: /3 ICON_CHECKED AS ICON,
              AT 9 SEL_JOBLIST-JOBNAME,
              AT 25 SEL_JOBLIST-JOBCOUNT.
  ELSEIF PRELIMINARY = 'X'.
    WRITE: /3 ICON_INCOMPLETE AS ICON,
              AT 9 SEL_JOBLIST-JOBNAME,
              AT 25 SEL_JOBLIST-JOBCOUNT.
  ELSEIF READY = 'X'.
    WRITE: /3 ICON_INCOMPLETE AS ICON,
              AT 9 SEL_JOBLIST-JOBNAME,
              AT 25 SEL_JOBLIST-JOBCOUNT.
  ELSEIF RUNNING = 'X'.
    WRITE: /3 ICON_INCOMPLETE AS ICON,
              AT 9 SEL_JOBLIST-JOBNAME,
              AT 25 SEL_JOBLIST-JOBCOUNT.
  ELSEIF SCHEDULED = 'X'.
    WRITE: /3 ICON_INCOMPLETE AS ICON,
              AT 9 SEL_JOBLIST-JOBNAME,
              AT 25 SEL_JOBLIST-JOBCOUNT.
  ENDIF.
ENDLOOP.
```

Selecting and Deleting a Job

To delete a background job explicitly, use:

- BP_JOB_SELECT to obtain the jobname and job number of the job that you wish to delete.

  You can select jobs according to all of the criteria available in the interactive background processing management system:

  – Jobname: Using a well-planned naming convention for your jobs will help you to select them precisely.
  – Job number
  – Name of the user who scheduled a job
  – Specifications for the start-time window/no start time scheduled
  – Start dependent upon predecessor jobs
  – Start dependent upon an event and event argument
  – Job status (preliminary, scheduled, ready, running, finished, aborted).

- BP_JOB_DELETE to delete the job. The job log is deleted as well, if the job has already been run.

  A job cannot delete itself. Also, a job that is currently running cannot be deleted. However, you can have it deleted automatically if it is completed successfully. See the DELANFREP parameter of JOB_OPEN.

Sample Program: Deleting a Background Job

```
* Data declarations:  BP_JOB_SELECT
*
DATA JSELECT LIKE BTCSELECT.

DATA SEL_JOBLIST LIKE TBTCJOB OCCURS 100 WITH HEADER LINE.

* Sample selection criteria
*
JSELECT-JOBNAME = 'Name of job'.
JSELECT-USERNAME = SY-UNAME.

  CALL FUNCTION 'BP_JOB_SELECT'
   EXPORTING
     JOBSELECT_DIALOG   = BTC_NO
     JOBSEL_PARAM_IN    = JSELECT
   IMPORTING
     JOBSEL_PARAM_OUT   = JSELECT
   TABLES
     JOBSELECT_JOBLIST  = SEL_JOBLIST
   EXCEPTIONS NO_JOBS_FOUND      =  1
     SELECTION_CANCELED =  2
     OTHERS             = 99.
```

```
*
* In this example, the program loops over the internal table
* SEL_JOBLIST and deletes each of the jobs that was selected.
*
* Alternative:  Have the user select the job to be deleted
* with BP_JOBLIST_PROCESSOR.  For an example, please see
* Sample Program:  Wait for Predecessor Job with JOB_CLOSE [Page 41].
*
  LOOP AT SEL_JOBLIST.
    CALL FUNCTION 'BP_JOB_DELETE'
     EXPORTING
       FORCEDMODE = 'X'
       JOBNAME    = SEL_JOBLIST-JOBNAME
       JOBCOUNT   = SEL_JOBLIST-JOBCOUNT
     EXCEPTIONS
       OTHERS     = 99.
  ENDLOOP.                                                      *
*
* FORCEDMODE deletes the job header even if other portions of the
* job cannot be deleted from the TemSe facility, where they are
* held.

* FORCEDMODE can be used without fear of causing problems in the
* System.  Any TemSe problem that affects background jobs can be
* resolved directly in the TemSe system and does not require the
* job header.
```

Using Events to Trigger Job Starts

Events let you start background jobs when particular changes in the R/3 System take place. When an event occurs, the background processing system starts all jobs that were scheduled to wait for that event.

The following sections explain event concepts and how to define and trigger events.

Event Concepts

Validity: Only in the Background Processing System

Events have meaning only in the background processing system. You can use events only to start background jobs.

Triggering an event notifies the background processing system that a named condition has been reached. The background processing system reacts by starting any jobs that were waiting for the event.

Types of Events

There are two types of events:

- System events are defined by SAP. These events are triggered automatically when such system changes as the activation of a new operation mode take place.
- User events are events that you define yourself. You must trigger these events yourself from ABAP or from external programs. You could, for example, signal the arrival of external data to be read into the R/3 System by using an external program to trigger a background processing event.

Event Arguments

You can qualify an event with an event argument. An event argument is a text string that you can optionally associate with an event. You can specify an event argument during the following operations:

- when you schedule a job to wait for the event
- when you trigger an event.

Unlike event IDs, event arguments are not defined in the R/3 System.

If you specify an argument when you schedule a job, then the job is eligible to start when the event is triggered. The job can start if:

- the event is triggered without any argument; or
- the event is triggered with the argument that you specified for the job.

If you do not specify an argument when you schedule a job, then the job can start as soon as the event occurs. The job is eligible to start no matter what argument string is supplied with the event.

When a Job Waiting for an Event May Start

Job	Event	Result
Job scheduled with Event ID “JSTART” Argument “A”	JSTART triggered, no argument	Job starts
	JSTART triggered with argument “A”	Job starts
	JSTART triggered with argument “B”	Job does not start; continues to wait for JSTART.

Job scheduled with Event ID “JSTART”, no argument	JSTART triggered, no argument	Job starts
	JSTART triggered with any argument	Job starts

An ABAP program that is running as a background job can find out what event and argument were presented when it was started. This makes it possible for ABAP programs running in the background to react intelligently to an event depending upon the argument string that was supplied with it.

Example: Events and Event Arguments

Switching to a different operation mode triggers an event in the background processing system. The event is an R/3 System event named SAP_OPMODE_SWITCH. As an argument, the event carries the name of the new operating mode.

If you schedule a job to wait upon the event SAP_OPMODE_SWITCH with argument NIGHT, then your job will become eligible to start when the operating mode NIGHT next becomes active.

You can also schedule jobs to be repeated whenever an event occurs. In the previous example, your job would be run whenever operating mode NIGHT becomes active, not just the first time that the event occurs.

You can schedule a job to wait for an operation mode to become active with the *Operation mode* button in the job scheduling function. You need not schedule the job to wait for the SAP_OPMODE_SWITCH event.

Using Events: Task Overview

You can use events that have already been defined, or you can create new events for scheduling background jobs.

If you wish to use new events, do the following to implement the event scheduling:

1. **Define and transport the event** as a user event with transaction SM62.

   You must define only event IDs; event arguments are not defined in the R/3 System. Instead, you specify event arguments when you schedule a job to wait for an event and when you trigger the event.

   If you define a new event, you must also transport it to your production systems. The event transaction does not have a connection to the transport system. Instead, you must create a transport request for the event yourself.

   Do this to transport an event:

   a. Create a transport request.

   b. Start the editor in the transport request and enter the following:

      `R3TR TABU <table name>` where table name is BTCSEV for a system event ID, BTCUEV for a user event ID.

   c. Press F2 with the cursor on the table name to call up the screen for specifying the table entries to transport. In this screen, enter the event ID's that you have created.

   d. Save and release the transport request. Ensure that it is imported into your production system(s).

2. To **trigger an event**, add:

   – the function module BP_EVENT_RAISE to your ABAP program, or

   – the program SAPEVT to your external script, batch file, or program.

   When your programs execute these keywords, an event will be triggered in the R/3 background processing system. The event-based scheduler is started immediately. It in turn starts all jobs that were waiting upon the event, subject to normal background processing restrictions, such as the requirement that the job has been released to start.

3. Schedule the jobs that are to run when your events are triggered.

   You can schedule jobs for one-time start or to be started whenever an event is triggered.

Defining Events

Use transaction SM62 to define and display events.

To define an event, do the following:

1. Enter transaction SM62.
2. Select *User event description*

   System events are reserved for SAP's use.
3. In the maintenance screen, enter the name of the event as it appears in your program and a brief description.

   You do not need to define event arguments.

Triggering Events from ABAP Programs

Use function module BP_EVENT_RAISE to trigger an event from an ABAP program.

Example

```
* Report processing before triggering event...
*
* Trigger event to start background jobs waiting for the event.
*
  DATA: EVENTID LIKE TBTCJOB-EVENTID.
  DATA: EVENTPARM LIKE TBTCJOB-EVENTPARM.

  EVENTID     = 'SP_TEST_EVENT'. " Event name must be defined
                                 " with transaction SM62.

  EVENTPARM   = 'EVENT1'.        " Optional:  a job can be
                                 " scheduled to wait for an
                                 " EVENTID or combination of
                                 " EVENTID and EVENTPARM.

  CALL FUNCTION 'BP_EVENT_RAISE' " Event is triggered. Jobs
   EXPORTING                     " waiting for event will be
     EVENTID    = EVENTID       " started.
     EVENTPARM  = EVENTPARM
     TARGET_INSTANCE = ' '       " Instance at which an event
                                 " should be processed.  Can
                                 " generally be omitted.
   EXCEPTIONS OTHERS = 1.        " Exceptions include event not
                                 " defined, no EVENTID
                                 " exported, etc.
```

Triggering Events from External Programs

Use the R/3 program SAPEVT to trigger an event from a program running outside the R/3 System.

The syntax for SAPEVT is as follows:

```
sapevt <EVENTID> [-p <EVENTPARM>] [-t]
pf=<Profile Name>|name=<R/3 System name> nr=<R/3 System Number>
```

Example: **sapevt END_OF_FI_DATATRANS name=C11 nr=11** triggers the event END_OF_FI_DATATRANS without a parameter. The event is triggered in R/3 System C11, instance number 11. Any jobs that are waiting for this event are started.

The SAPEVT parameters are as follows:

- **<EVENTID>**: The name of the event as defined in the R/3 System with transaction SM62.
- **-p <EVENTPARM>**: An optional argument for the EVENTID which qualifies the event. The EVENTPARM is not defined in the R/3 System.
- **-t**: Causes SAPEVT to log its actions in a short trace file. You can find the trace file in the current directory of the user who called SAPEVT.
- **pf=<Profile Name>** or **name=<R/3 System Name> nr=<R/3 System Number>**: Either pf or name and nr identify the R/3 System in which the event is to be triggered.

  At least one instance of the R/3 System must be active when SAPEVT runs. Otherwise, the System will not know that the event has occurred. The instance must have the number specified in the SAPEVT call.

  In more detail, the parameters for identifying the R/3 System are as follows:

  – **pf <Profile Name>**: Enter the name of the profile with which the R/3 background processing instance(s) are started. SAPEVT identifies the R/3 System from the specifications in the profile.

    You can find R/3 system profiles in the globally shared R/3 directory SYS\profile (Windows NT), or SYS/profile (UNIX).

  – **name=<R/3 System Name> nr=<R/3 System Number>**: The name of the R/3 System (SID) and the R/3 System number.

Requirements for Using SAPEVT

To use SAPEVT to trigger an event, the following requirements must be met:

- An R/3 instance that offers background processing must be running or the start of jobs waiting on the event will be delayed.

  If no instance in an R/3 System is active when an external program triggers an event, than the event is ignored, the R/3 System will not know that the event has occurred.

  If a job that was scheduled to run when an event occurs cannot be started immediately, it is rescheduled as an immediate-start job. It will then be run at the earliest possible opportunity.

Triggering Events from External Programs

- If SAPEVT is started with a profile name as an argument, then the program must have access to the shared `profile` directory of the R/3 System. SAPEVT needs to read the profile of the system to which it is sending the event.

  If access to the directory is not possible, then you can create a local profile for SAPEVT. This profile need only contain values for the parameters *rdisp/mshost* and *rdisp/msserv*. Take the values from the profiles of the target R/3 System. Specify the local profile in the SAPEVT command line.

- Network connection: SAPEVT must be able to establish a TCP connection to the R/3 instance.

  SAPEVT reports problems in establishing a connection to the R/3 System in its log file.

- User search path: The R/3 executables and profile directories must lie in the search path of the user who starts the program that calls SAPEVT.

  UNIX: /usr/sap/<R/3 System ID>/SYS/exe/run and SYS/profile

  Windows NT: \\<Host Name>\sapmnt\<R/3 System ID>\SYS\exe\run and SYS\profile

Finding Out Which Event and Argument Were Triggered

As of Release 2.2B, you can use the function module GET_JOB_RUNTIME_INFO, you can determine what event and argument triggered the start of a background job from within the background job. This is possible only in job steps that start ABAP programs.

Finding out how a job was triggered makes it possible for a background job to react intelligently to the event with which it was started. Example: You schedule a job that is to be started whenever the event FI_DATAIMPORT_DONE (some phase of a data import has been completed) is triggered, without specifying any event argument. When the job is started, you can determine from within the job which event argument was provided with the event. In this example, your job could therefore find out which phase of the data import was just completed. Depending upon whether the event argument was "FI_TABLE1 " or "FI_TABLE2", your job could respond differently.

Running External Programs

You can use a background job to start an external program, that is, any executable file on any host system that is accessible from the R/3 System.

The following sections describe the requirements for starting external programs and special techniques for working with external programs.

You can find an example of the JOB_SUBMIT call for starting an external program in Sample Program: Adding a Job Step for an External Command or Program [Page 31].

Requirements

You can start external programs from a background job only if the following system requirements are met:

- Gateway required: The R/3 gateway server must be running. It must be able to establish a CPI-C connection between the background processing server and the host system on which the external program is to be started.
- Search path: You must specify the full pathname of the external program in your ABAP background job, or the external program must lie in the search path of the R/3 CPI-C user.

  The gateway server starts the program with the CPI-C user. Usually, this is the R/3 host system user <R/3 System Name>adm (c11adm, for example).
- R/3 executables in search path: The R/3 "executables" directory must lie in the search path of the R/3 CPI-C user. The external program is started by an R/3 control program, which also receives the return code issued by the external program.

  The executables directory is as follows:

  – UNIX: /usr/sap/<R/3 System ID>/SYS/exe/run
  – Windows NT: \\<Host Name>\sapmnt\<R/3 System ID>\SYS\exe\run

Special Techniques: Starting External Programs [Page 76]

Implementing Parallel Processing

For some R/3 reports, the nights are getting too short. Especially at customers with large volumes of data, some R/3 reports that customarily run in the background processing system (such as material planning runs) may have run times of many hours. It can be difficult to finish such jobs in the “night-time” that is available, especially if dialog users are spread across several time zones.

With Release 3.1G, R/3 offers a solution to the “short nights” problem: parallel-processed background jobs. Long-running R/3 reports can now implement parallel processing, which lets them parcel out the work to be done to available dialog work processes in the R/3 System and then collect the results.

Parallel processing is implemented in ABAP reports and programs, not in the background processing system itself. That means that jobs are only processed in parallel if the report that runs in a job step is programmed for parallel processing. Such reports can also process in parallel if they are started interactively.

Parallel-processing is implemented with a special variant of asynchonous RFC. It’s important that you use only the correct variant for your own parallel processing applications: the CALL FUNCTION STARTING NEW TASK DESTINATION IN GROUP keyword. Using other variants of asynchronous RFC circumvents the built-in safeguards in the correct keyword, and can bring your system to its knees

Function Modules and ABAP Keywords

Parallel processing is implemented in the application reports that are to run in the background. You can implement parallel processing in your own background applications by using the following function modules and ABAP keywords:

- Function module SPBT_INITIALIZE: Optional. Use to determine the availability of resources for parallel processing.

  You can do the following:

  – check that the parallel processing group that you have specified is correct.

  – find out how many work processes are available so that you can more efficiently size the packets of data that are to be processed in your data.

- ABAP keyword CALL FUNCTION <function> STARTING NEW TASK <taskname> with the DESTINATION IN GROUP argument: Use this keyword to have the R/3 System execute the function module call in parallel. Typically, you’ll place this keyword in a loop in which you divide up the data that is to be processed into work packets. You can pass the data that is to be processed in the form of an internal table (EXPORT, TABLE arguments). The keyword implements parallel processing by dispatching asynchronous RFC calls to the servers that are available in the RFC server group specified for the processing.

  Note that your RFC calls with CALL FUNCTION are processed in work processes of type DIALOG. The DIALOG limit on processing of one dialog step (by default 300 seconds, system profile parameter rdisp/max_wprun_time) applies to these RFC calls. Keep this limit in mind when you divide up data for parallel processing calls.

- Function module SPBT_GET_PP_DESTINATION: Optional. Call immediately after the CALL FUNCTION keyword to get the name of the server on which the parallel processing task will be run.
- Function module SPBT_DO_NOT_USE_SERVER: Optional. Excludes a particular server from further use for processing parallel processing tasks. Use in conjunction with SPBT_GET_PP_DESTINATION if you determine that a particular server is not available for parallel processing (for example, COMMUNICATION FAILURE exception if a server becomes unavailable).
- ABAP keyword WAIT: Required if you wish to wait for all of the asynchronous parallel tasks created with CALL FUNCTION to return. This is normally a requirement for orderly background processing. May be used only if the CALL FUNCTION includes the PERFORMING ON RETURN addition.
- ABAP keyword RECEIVE: Required if you wish to receive the results of the processing of an asynchronous RFC. RECEIVE retrieves IMPORT and TABLE parameters as well as messages and return codes.

Prerequisites

Before you implement parallel processing, make sure that your background processing application and your R/3 System meet these requirements:

- Logically-independent units of work: The data processing task that is to be carried out in parallel must be logically independent of other instances of the task. That is, the task can be carried out without reference to other records from the same data set that are also being processed in parallel, and the task is not dependent upon the results of others of the parallel operations. For example, parallel processing is not suitable for data that must be sequentially processed or in which the processing of one data item is dependent upon the processing of another item of the data.
- By definition, there is no guarantee that data will be processed in a particular order in parallel processing or that a particular result will be available at a given point in processing.
- ABAP requirements (see also the online documentation for the CALL FUNCTION STARTING NEW TASK DESTINATION IN GROUP keyword):
  - The function module that you call must be marked as externally callable. This attribute is specified in the *Remote function call supported* field in the function module definition (transaction SE37)).
  - The called function module may not include a function call to the destination “BACK.”
  - The calling program should not change to a new internal session after making an asynchronous RFC call. That is, you should not use SUBMIT or CALL TRANSACTION in such a report after using CALL FUNCTION STARTING NEW TASK.
  - You cannot use the CALL FUNCTION STARTING NEW TASK DESTINATION IN GROUP keyword to start external programs.
  - In calls between R/3 Systems, both systems must be of Release 3.0A or higher.
- R/3 System resources: In order to process tasks from parallel jobs, a server in your R/3 System must have at least 3 dialog work processes. It must also meet the workload criteria of the parallel processing system: Dispatcher queue less than 10% full, at least one dialog work process free for processing tasks from the parallel job.

Managing Resources with RFC Server Groups

The parallel processing system has built-in safeguards that eliminate the possibility that a parallel job can soak up all of the resources in an R/3 System and cause performance problems for other jobs or other users.

In addition to these built-in safeguards, you can optimize the sharing of resources through RFC server groups. In the context of parallel processing, a group specifies the set of R/3 application servers that can be used for executing a particular program in parallel. By default (CALL FUNCTION STARTING NEW TASK with the addition DESTINATION IN GROUP DEFAULT), the group is all servers that meet the resource criteria. But you can also create your own more limited groups. You can view and maintain groups with transaction RZ12 (*Tools → Administration → Administration → Network → RFC destinations* and then *RFC → RFC groups*).

You must specify the group to use in both the SPBT_INITIALIZE function module (if used) and in the ABAP CALL FUNCTION STARTING NEW TASK keyword. Only one group is allowed per parallel ABAP report or program (job step).

Messages and Exceptions

In the function modules that you call, you should use exceptions for any error reporting, and not the MESSAGE keyword. Exception handling is fully under your control in asynchronous RFC. You simply add the exceptions generated by the function module to the reserved SYSTEM_FAILURE and COMMUNICATIONS_FAILURE exceptions of the CALL FUNCTION keyword. You can then handle the exceptions in the program that launches the parallel programming tasks.

Authorizations

If the system profile parameter auth/rfc_authority_check is set (value 1), then the System automatically checks at the CALL FUNCTION keyword whether the authorizations user of the background job has the required RFC authorization. The RFC authorization object is S_*RFC Authorization check at RFC access.* The authorization checks access to function modules by function module group. That is, whether a user has the right to run function modules that belong to a particular group.

You can test a user's RFC authorization with the function module AUTHORITY_CHECK_RFC. This function module returns RC = 0 if the user is authorized for the group that you name. The function module does not check whether an authority check will actually take place.

Sample Program: Parallel Job

This sample program shows how to create a report that would execute in parallel if started interactively or in the background processing system. It is based upon the online documentation for the ABAP CALL FUNCTION STARTING NEW TASK documentation.

Program Structure

SPBT_INITIALIZE: After declaring data, the report calls the SPBT_INITIALIZE function module. The call lets the program check that the parallel processing group is valid and that resources are available. This call is optional. If you do not call the function module, then the R/3 System itself calls SPBT_INITIALIZE to initialize the RFC server group.

Since the function module returns the number of available work processes, the call could be used to decide how to size the work packets that are to be processed. If 10 work processes were

available, you could, for example, divide the data into 10 packets for processing. However, there's no guarantee that the number of free work processes will not change between the time of the call and the time that your program sends off its work packets unless you are working with a group of servers that are reserved for your job.

It's of course also possible to run through the data making one parallel processing call for each record that is to be processed. In this case, of course, no attempt is made to optimize the sizing of work packets.

CALL FUNCTION... Loop: The heart of the report is a DO loop in which the function module that is to be processed in parallel is called (CALL FUNCTION STARTING NEW TASK DESTINATION IN GROUP). The loop in this example is controlled by a simple count-down mechanism. In a production report, you'd repeat the loop until all of your data has been sent off for processing with CALL FUNCTION.

For purposes of simplicity, the loop in this example calls a function module that requires no data (RFC_SYSTEM_INFO). In a production report, you'd include logic to select a record or group of records for processing. You'd package these recorts in an internal table and include them with the parallel processing call using the EXPORTING or TABLES additions of CALL FUNCTION STARTING NEW TASK.

For purposes of recovery, a production program should also include logic for logging the progress of processing. Should the program terminate abnormally in the middle of processing, it's essential that you be able to determine which data has already been processed and with which data the report should resume. In simplest form, this logic should log the data that has been dispatched for parallel processing and should note the completion of the processing of each unit of data.

Task management: A 0 return code (SY-SUBRC) from CALL FUNCTION indicates that your parallel processing task has been successfully dispatched. It's now important that your job keep track of these parallel processing tasks. Your task management should take care of two assignments:

- Generating unique task names for each CALL FUNCTION task.

  Use the taskname that you specified in CALL FUNCTION to identify parallel processing tasks. Each such task that you dispatch should have a unique name so that you can correctly identify returns from the task.

- Waiting for resources to become available, should all parallel processing resources (dialog work processes) temporarily be in use. See "Handling the CURRENTLY_NO_RESOURCES_AVAIL exception," below.

- Determining when all tasks have been completed. Only then can your program terminate. See "Waiting for job completion", below, for more information.

In the example, the task management increments a counter as the task name and maintains a table of the tasks that have been dispatched. As an optional feature, the task management uses SPBT_GET_PP_DESTINATION to find out where each parallel task is being processed.

Handling the RESOURCE_FAILURE exception: As each parallel processing task is dispatched, the R/3 System counts down the number of resources (dialog work processes) available for processing additional tasks. This count goes up again as each parallel processing task is completed and returns to your program.

Should your parallel processing tasks take a long time to complete, then the parallel processing resources may temporarily run out. In this case, CALL FUNCTION returns the exception

RESOURCE_FAILURE. This means simply that all dialog work processes in the RFC group that your program is using are in use.

Your program must now wait until resources become available and then re-issue the CALL FUNCTION that failed. In the sample program, we use a simple, reasonably failsafe wait mechanism. The program waits for parallel processing tasks to return, freeing up resources. The WAIT also specifies a initial timeout of 1 second. If the CALL FUNCTION again fails, the WAIT is repeated with a longer time-out. You can increase the time-outs if you expect that your parallel tasks will take longer to complete. You should also add code to exit from the retry loop after a suitable number of iterations.

Receiving replies: The CALL FUNCTION keyword triggers processing of the form RETURN_INFO when each parallel processing task completes. This form uses the RECEIVE keyword to capture the results of the parallel processing. In this case, the structure RFCSI_EXPORT from RFC_SYSTEM_INFO is collected into the internal structure INFO.

RECEIVE is needed to gather IMPORTING and TABLE returns of an asynchronously executed RFC function module.

Example: Assume that your report generates a list. You would use a form like RETURN_INFO and RECEIVE to collect the list entries generated by each parallel processing task. After all parallel tasks have returned, your report could then sort or otherwise further process the list entries before presenting them.

Waiting for job completion: As part of your task management, your job must wait until all of the parallel processing tasks have been completed. To do this, the sample program uses the WAIT keyword to wait until the number of completed parallel processing tasks is equal to the number of tasks that were created. Independently of this WAIT; the RETURN_INFO form is triggered. RETURN_INFO keeps track of the number of completed parallel processing tasks, so that the WAIT condition can be correctly evaluated.

```
REPORT PARAJOB.
*
* Data declarations
*
DATA: GROUP LIKE RZLLITAB-CLASSNAME VALUE ' ',
                                "Parallel processing group.
                                "SPACE = group default (all
                                "servers)
      WP_AVAILABLE TYPE I,      "Number of dialog work processes
                                "available for parallel processing
                                "(free work processes)
      WP_TOTAL TYPE I,          "Total number of dialog work
                                "processes in the group
      MSG(80) VALUE SPACE,      "Container for error message in
                                "case of remote RFC exception.
      INFO LIKE RFCSI, C,       "Message text
      JOBS TYPE I VALUE 10,     "Number of parallel jobs
      SND_JOBS TYPE I VALUE 1,  "Work packets sent for processing
      RCV_JOBS TYPE I VALUE 1,  "Work packet replies received
      EXCP_FLAG(1) TYPE C,      "Number of RESOURCE_FAILUREs
      TASKNAME(4) TYPE N VALUE '0001',  "Task name (name of
                                "parallel processing work unit)
      BEGIN OF TASKLIST OCCURS 10,  "Task administration
         TASKNAME(4) TYPE C,
         RFCDEST     LIKE RFCSI-RFCDEST,
```

```
         RFCHOST     LIKE RFCSI-RFCHOST,
       END OF TASKLIST.
*
* Optional call to SBPT_INITIALIZE to check the
* group in which parallel processing is to take place.
* Could be used to optimize sizing of work packets
* work / WP_AVAILABLE).
*
CALL FUNCTION 'SPBT_INITIALIZE'
  EXPORTING
    GROUP_NAME                    = GROUP
                                    "Name of group to check
  IMPORTING
    MAX_PBT_WPS                   = WP_TOTAL
                                    "Total number of dialog work
                                    "processes available in group
                                    "for parallel processing
    FREE_PBT_WPS                  = WP_AVAILABLE
                                    "Number of work processes
                                    "available in group for
                                    "parallel processing at this
                                    "moment
  EXCEPTIONS
    INVALID_GROUP_NAME            = 1
                                    "Incorrect group name; RFC
                                    "group not defined. See
                                    "transaction RZ12
    INTERNAL_ERROR                = 2
                                    "R/3 System error; see the
                                    "system log (transaction
                                    "SM21) for diagnostic info
    PBT_ENV_ALREADY_INITIALIZED   = 3
                                    "Function module may be
                                    "called only once; is called
                                    "automatically by R/3 if you
                                    "do not call before starting
                                    "parallel processing
    CURRENTLY_NO_RESOURCES_AVAIL = 4
                                    "No dialog work processes
                                    "in the group are available;
                                    "they are busy or server load
                                    "is too high
    NO_PBT_RESOURCES_FOUND        = 5
                                    "No servers in the group
                                    "met the criteria of >
                                    "two work processes
                                    "defined.
    CANT_INIT_DIFFERENT_PBT_GROUPS = 6
                                    "You have already initialized
                                    "one group and have now tried
                                    "initialize a different group.
    OTHERS                         = 7..
```

```
CASE SY-SUBRC.
    WHEN 0.
       "Everything's ok. Optionally set up for optimizing size of
       "work packets.
    WHEN 1.
       "Non-existent group name.  Stop report.
       MESSAGE E836. "Group not defined.
    WHEN 2.
       "System error.  Stop and check system log for error
       "analysis.
    WHEN 3.
       "Programming error.  Stop and correct program.
       MESSAGE E833. "PBT environment was already initialized.
    WHEN 4.
       "No resources: this may be a temporary problem.  You
       "may wish to pause briefly and repeat the call.  Otherwise
       "check your RFC group administration:  Group defined
       "in accordance with your requirements?
       MESSAGE E837. "All servers currently busy.
    WHEN 5.
       "Check your servers, network, operation modes.
    WHEN 6.

* Do parallel processing.  Use CALL FUNCTION STARTING NEW TASK
* DESTINATION IN GROUP to call the function module that does the
* work.  Make a call for each record that is to be processed, or
* divide the records into work packets.  In each case, provide the
* set of records as an internal table in the CALL FUNCTION
* keyword (EXPORT, TABLES arguments).

DO.
  CALL FUNCTION 'RFC_SYSTEM_INFO'     "Function module to perform
                                      "in parallel
       STARTING NEW TASK TASKNAME     "Name for identifying this
                                      "RFC call
       DESTINATION IN GROUP group     "Name of group of servers to
                                      "use for parallel processing.
                                      "Enter group name exactly
                                      "as it appears in transaction
                                      "RZ12 (all caps).  You may
                                      "use only one group name in a
                                      "particular ABAP program.
       PERFORMING RETURN_INFO ON END OF TASK
                                      "This form is called when the
                                      "RFC call completes. It can
                                      "collect IMPORT and TABLES
                                      "parameters from the called
                                      "function with RECEIVE.
       EXCEPTIONS
         COMMUNICATION_FAILURE = 1 MESSAGE msg
                                      "Destination server not
```

```
                                    "reached or communication
                                    "interrupted.  MESSAGE msg

                                    "captures any message
                                    "returned with this
                                    "exception (E or A messages
                                    "from the called FM, for
                                    "example.  After exception
                                    "1 or 2, instead of aborting
                                    "your program, you could use
                                    "SPBT_GET_PP_DESTINATION and
                                    "SPBT_DO_NOT_USE_SERVER to
                                    "exclude this server from
                                    "further parallel processing.
                                    "You could then re-try this
                                    "call using a different
                                    "server.
        SYSTEM_FAILURE        = 2  MESSAGE msg
                                    "Program or other internal
                                    "R/3 error.  MESSAGE msg
                                    "captures any message
                                    "returned with this
                                    "exception.
        RESOURCE_FAILURE      = 3.  "No work processes are
                                    "currently available.  Your
                                    "program MUST handle this
                                    "exception.
        YOUR_EXCEPTIONS        = X. "Add exceptions generated by
                                    "the called function module
                                    "here.  Exceptions are
                                    "returned to you and you can
                                    "respond to them here.

CASE SY-SUBRC.
  WHEN 0.
    "Administration of asynchronous RFC tasks
    "Save name of task...
    TASKLIST-TASKNAME = TASKNAME.
    "... and get server that is performing RFC call.
    CALL FUNCTION 'SPBT_GET_PP_DESTINATION'
      EXPORTING
        RFCDEST = TASKLIST-RFCDEST
      EXCEPTIONS
        OTHERS  = 1.
    APPEND TASKLIST.
    WRITE: /  'Started task: ', TASKLIST-TASKNAME COLOR 2.

    TASKNAME = TASKNAME + 1.
    SND_JOBS = SND_JOBS + 1.
    "Mechanism for determining when to leave the loop.  Here, a
    "simple counter of the number of parallel processing tasks.
    "In production use, you would end the loop when you have
    "finished dispatching the data that is to be processed.
```

```
  JOBS     = JOBS - 1.  "Number of existing jobs
  IF JOBS = 0.
    EXIT.  "Job processing finished

  ENDIF.
WHEN 1 OR 2.
  "Handle communication and system failure.  Your program must
  "catch these exceptions and arrange for a recoverable
  "termination of the background processing job.
  "Recommendation:  Log the data that has been processed when
  "an RFC task is started and when it returns, so that the
  "job can be restarted with unprocessed data.
  WRITE msg.
  "Remove server from further consideration for
  "parallel processing tasks in this program.
  "Get name of server just called...
  CALL FUNCTION 'SPBT_GET_PP_DESTINATION'
    EXPORTING
      RFCDEST = TASKLIST-RFCDEST
    EXCEPTIONS
      OTHERS  = 1.
  "Then remove from list of available servers.
  CALL FUNCTION 'SPBT_DO_NOT_USE_SERVER'
    IMPORTING
      SERVERNAME = TASKLIST-RFCDEST
    EXCEPTIONS
      INVALID_SERVER_NAME         = 1
      NO_MORE_RESOURCES_LEFT      = 2
                                    "No servers left in group.
      PBT_ENV_NOT_INITIALIZED_YET = 3
      OTHERS                      = 4.
WHEN 3.
  "No resources (dialog work processes) available at
  "present.  You need to handle this exception, waiting
  "and repeating the CALL FUNCTION until processing
  "can continue or it is apparent that there is a
  "problem that prevents continuation.
  MESSAGE I837. "All servers currently busy.
  "Wait for replies to asynchronous RFC calls.  Each
  "reply should make a dialog work process available again.
  IF EXCP_FLAG = SPACE.
     EXCP_FLAG = 'X'.
     "First attempt at RESOURCE_FAILURE handling.  Wait
     "until all RFC calls have returned or up to 1 second.
     "Then repeat CALL FUNCTION.
     WAIT UNTIL RCV_JOBS >= SND_JOBS UP TO '1' SECONDS.
  ELSE.
     "Second attempt at RESOURCE_FAILURE handling
     WAIT UNTIL RCV_JOBS >= SND_JOBS UP TO '5' SECONDS.
     "SY-SUBRC 0 from WAIT shows that replies have returned.
     "The resource problem was therefore probably temporary
     "and due to the workload.  A non-zero RC suggests that
     "no RFC calls have been completed, and there may be
```

```
          "problems.
          IF SY-SUBRC = 0.
            CLEAR EXCP_FLAG.
          ELSE.  "No replies

            "Endless loop handling
          ...
          ENDIF.
        ENDIF.
    ENDCASE.
ENDDO.
...
*
* Wait for end of job:  replies from all RFC tasks.
* Receive remaining asynchronous replies
WAIT UNTIL RCV_JOBS >= SND_JOBS.
LOOP AT TASKLIST.
  WRITE:/   'Received task:', TASKLIST-TASKNAME COLOR 1,
         30  'Destination: ', TASKLIST-RFCDEST COLOR 1.
ENDLOOP.
...
*
* This routine is triggered when an RFC call completes and
* returns.  The routine uses RECEIVE to collect IMPORT and TABLE
* data from the RFC function module.
*
* Note that the WRITE keyword is not supported in asynchronous
* RFC.  If you need to generate a list, then your RFC function
* module should return the list data in an internal table.  You
* can then collect this data and output the list at the conclusion
* of processing.
*
FORM RETURN_INFO USING TASKNAME.

   DATA:  INFO_RFCDEST LIKE TASKLIST-RFCDEST.

   RECEIVE RESULTS FROM FUNCTION 'RFC_SYSTEM_INFO'
     IMPORTING RFCSI_EXPORT = INFO
     EXCEPTIONS
       COMMUNICATION_FAILURE = 1
       SYSTEM_FAILURE        = 2.

   RCV_JOBS = RCV_JOBS + 1.  "Receiving data
     IF SY-SUBRC NE 0.
     * Handle communication and system failure
    ...
```

```
ELSE.
       READ TABLE TASKLIST WITH KEY TASKNAME = TASKNAME.
       IF SY-SUBRC = 0.  "Register data
         TASKLIST-RFCHOST = INFO_RFCHOST.
         MODIFY TASKLIST INDEX SY-TABIX.
       ENDIF.
     ENDIF.
...
ENDFORM
```

Special Techniques: Starting External Programs

This section describes special procedures to use for starting:

- service programs or daemons that are to remain active in the target system
- multiple programs in cascade fashion

Starting Service Programs and Daemons

You can use the R/3 background processing system to start service programs, such as daemons in UNIX systems.

These programs are intended to remain active after they have been started. They do not terminate and return to the R/3 background control program like normal programs.

When you start a service program, you should use the following *Control flag* setting when you schedule the job:

Termination: don't wait: Since the external program should not immediately terminate, you should change the default Wait setting. The R/3 control program will then terminate as soon as it has started the external program.

You can still get trace data back from the control program up until the time that it starts the external program.

Starting Cascades of Programs

Often, an external program started by the background processing system in turn starts one or more other external programs. This cascading of programs presents a problem: often, you will not be able to obtain useful feedback from the initial external program. If the external program terminates without waiting for the other programs to complete, then the return code in the job log provides no useful information on the additional processing done in the target system.

You can avoid this problem in two ways:

- If you can modify the first external program, then have it issue an event to the R/3 background processing system rather than starting the additional programs.

  In the background processing system, you can then schedule individual jobs for each of the "cascaded" programs that are to be started. When the initial program issues the event, the background processing system starts each of the jobs that is waiting for the event.

  Advantage: you can receive return codes, output, and/or trace data from each of the external programs in the job logs.

- If you cannot modify a "master program," then modify the "cascaded" programs to signal their outcomes to the R/3 system with events. You can use the event to have the R/3 background processing system start additional processing.

Example: during data transfer into your R/3 System, you could start each external data transfer program from a script or small control program that waits for the transfer program to terminate.

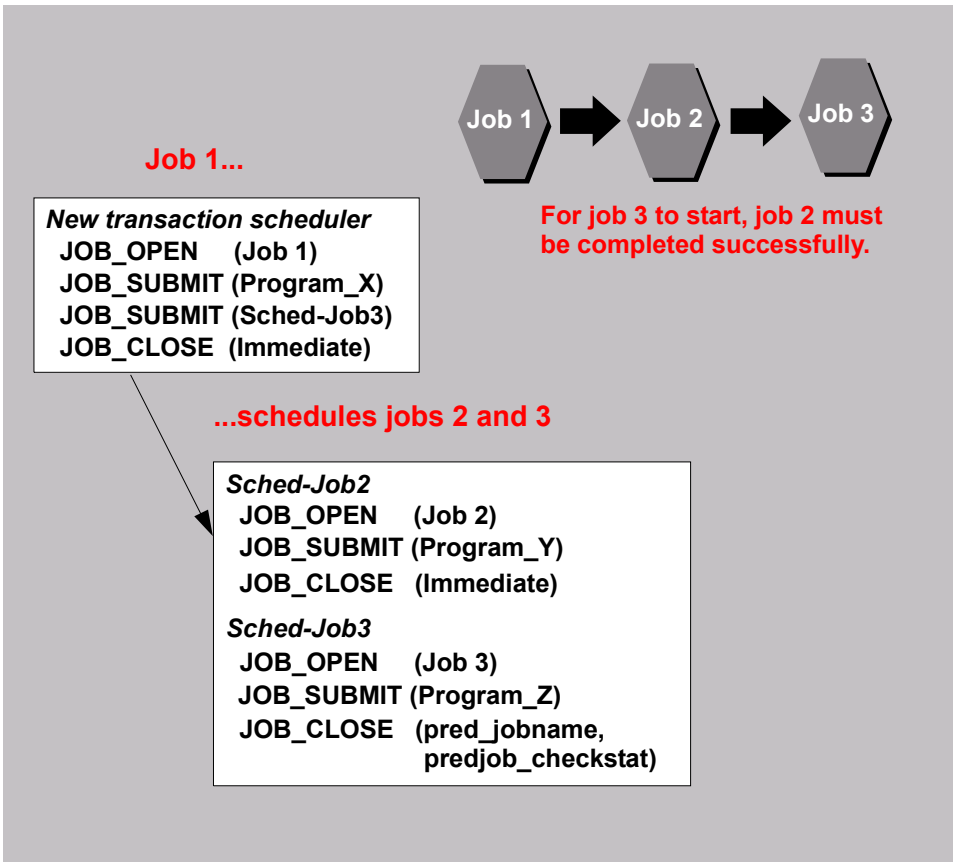
Upon termination of the transfer program, the control program checks the return code and issues the appropriate event to the R/3 system with the R/3 program SAPEVT. You can have the R/3 background processing system use the event to start the ABAP program that processes the transferred data.

Advanced Scheduling 1: Building Chains of Jobs

Goal: Assume that you irregularly have to schedule a chain of jobs, that is, a set of jobs linked by predecessor-successor relationships.

You'd like to consolidate the scheduling of this chain of jobs into a single job that sees to the scheduling of all jobs in the chain. The advantage of this consolidation: you need only to schedule a single job to launch the chain of jobs.

Scheduling a Job Chain.



Implementation: You'd implement this job in three small ABAP reports. These show both:

- a loose, informal predecessor-successor relationship (between Jobs 1 and 2, no guarantee that Job 1 completes before Job 2); and
- a tight relationship using the "start after predecessor" mechanism of the background processing system (between jobs 2 and 3). Here, there is a guarantee that Job 2 finishes before Job 3. In this case, there's even a guarantee that Job 2 has completed successfully before Job 3 can start.

The first report schedules Job 1 to run immediately. Job 1 does some work and schedules both Job 2 and Job 3. There is no formal predecessor/successor relationship between Jobs 1 and 2. Usually, Job 2 won't start until Job 1 processing has been completed. But it's possible for steps

in Jobs 1 and 2 to overlap. For example, if “Program_X” in Job 1 is an external program and Job 1 does not wait for Program_X to complete, then “Program_Y” in Job 2 and Program_X in Job 1 could overlap.

Job 1 must schedule both Jobs 2 and 3 because a job cannot schedule its own successor. This is because a predecessor job must have the status *Scheduled* or *Released* when a successor job is scheduled. If you want to set up chains of jobs, do so from a single initial report or job, as in this example.

Job 2, in turn, runs to completion and triggers the start of Job 3.. Here, Job 3 is formally linked to Job 2 as its predecessor, using the pred_jobname argument of the JOB_CLOSE function module. The predjob_checkstat argument ensures that Job 3 will start only if Job 2 is successfully completed. For a programming sample, see Sample Program: Wait for Predecessor Job with JOB_CLOSE [Page 41].

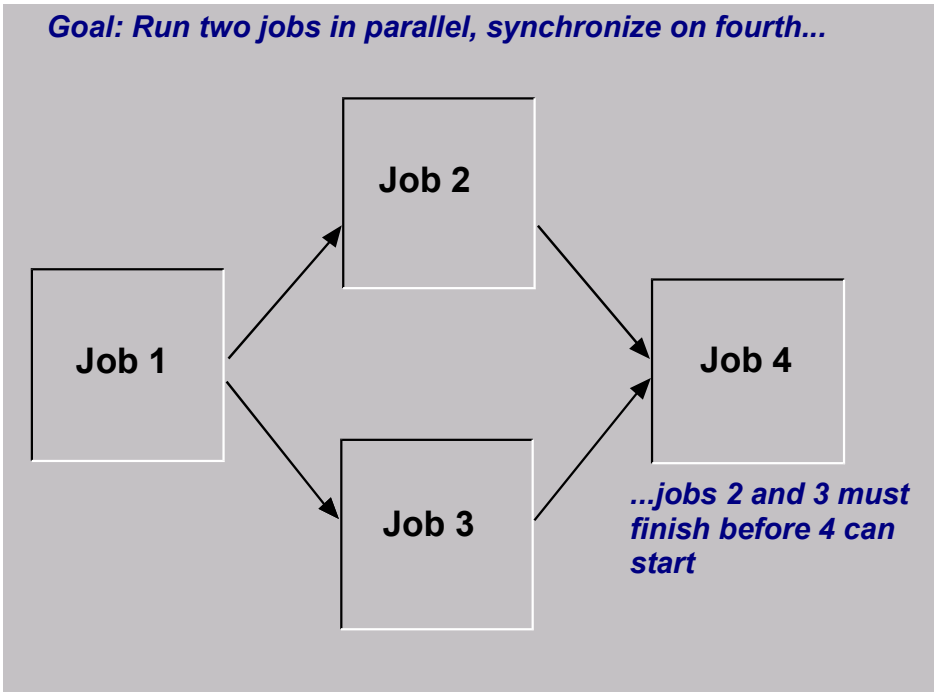
(An alternative to the strategy shown here would be to schedule the jobs in the chain to run periodically. But this is a good solution only if the jobs must run according to a regular schedule. Also, the jobs are not automatically rescheduled if one of them fails.)

Advanced Scheduling 2: Scheduling and Synchronizing Jobs in Parallel

Goal: Assume that you have the constellation shown in the diagram below: one job should launch two additional programs that run in parallel. When both of these are done, a fourth job should start.

You'd like to implement this consolidation in a single job. The advantage: you only have to start the master job to start the whole process.

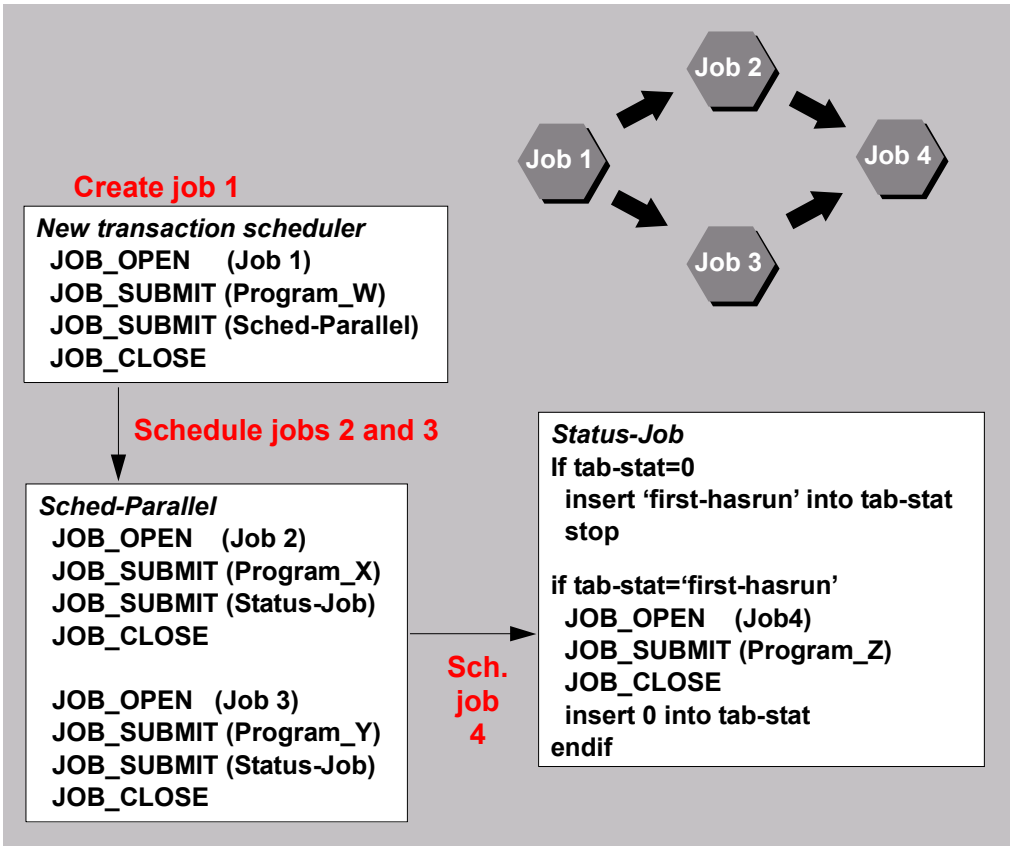
Synchronized Parallel Jobs



Implementation: The diagram below shows how this process could be implemented. An initial report schedules Job 1, which does preparatory work for Jobs 2 and 3 and then schedules them. (You could guarantee that Jobs 2 and 3 start after Job 1 with "start after predecessor" scheduling -- building a job chain between job1 and jobs 2 and 3 with a job or report that ran previously. For more information, see Advanced Scheduling 1: Building Chains of Jobs [Page 77]).

Implementing Parallel Synchronized Jobs

Advanced Scheduling 2: Scheduling and Synchronizing Jobs in Parallel

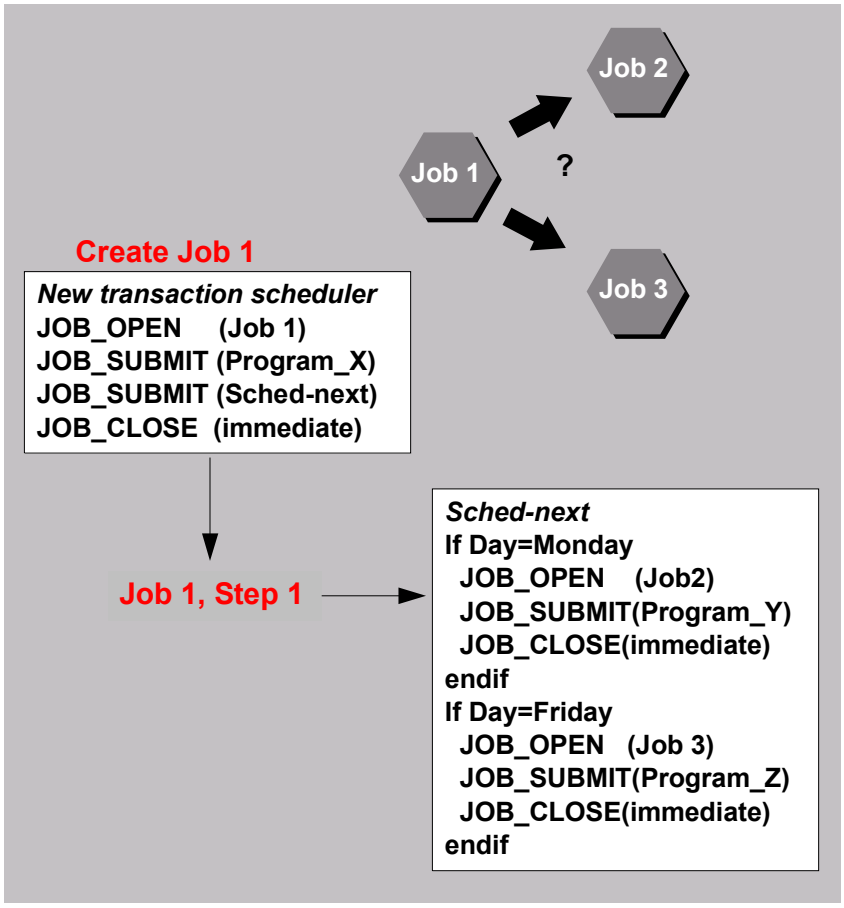


The synchronizing mechanism is provided by a small program that updates a status flag shared by Jobs 2 and 3. The first job sets the status and quits. The second job to run the status program schedules job 4.

Advanced Scheduling 3: Deciding Which Job to Schedule

Goal: Add decision-making logic to a background job. Very few procedures can be represented without decision-making logic. In this sample, the initial job implements some simple decision-making logic to decide which additional job to schedule. It does this by running the Sched-next program to evaluate the day of the week and schedule a job accordingly.

Implementing Simple Decision-Making.



Reference: Background Processing Function Modules

The following function modules for background processing are available. Do not use other function modules that belong to the background processing system; these are reserved for internal use.

JOB_OPEN: Create a Background Processing Job

Use JOB_OPEN to create a background job. The function module returns the unique ID number which, together with the job name, is required for identifying the job.

Once you have "opened" a job, you can add job steps to it with JOB_SUBMIT and submit the job for processing with JOB_CLOSE.

For more information, please see the online documentation in the function module facility (transaction SE37) or Sample Program: Creating a Job with JOB_OPEN [Page 29].

JOB_SUBMIT, ABAP SUBMIT: Add a Job Step to a Job

Use JOB_SUBMIT to add a job step to a background job that you have opened with JOB_OPEN.

A job step is an independent unit of work in a job, the execution of an ABAP or external program. Each job step can have its own authorizations user and printer/optical archiving specifications.

Related function modules include:

- JOB_OPEN: Create a background job.
- JOB_CLOSE: Submit a background job to the background processing system for execution.

For more information, please see the online documentation in the function module facility (transaction SE37) or Scheduling a Job: Full-Control Method [Page 17].

JOB_CLOSE: Pass a Job to the Background Processing System

Use JOB_CLOSE to pass a background job to the background processing system to be run. Once you have "closed" a job, you can no longer add job steps to it or change job/job step specifications.

The function module returns an indicator as to whether the job was automatically released or not. A job is automatically released to run only if the user who scheduled the job has RELE release authorization for the authorization object Operations on background jobs.

A job step is an independent unit of work in a job, the execution of an ABAP or external program. Each job step can have its own authorizations user and printer/optical archiving specifications.

Related function modules include:

- JOB_OPEN: Create a background job.
- JOB_SUBMIT: Add job steps to a job.

For more information, please see the online documentation in the function module facility (transaction SE37) or Scheduling a Job: Full-Control Method [Page 17].

BP_JOBVARIANT_SCHEDULE and BP_JOBVARIANT_OVERVIEW: Easy Job Scheduling and Management

These two function modules offer an “easy-method” or “express-method” for scheduling and then managing background processing jobs.

The function modules are as follows:

- BP_JOBVARIANT_SCHEDULE [Page 13]: Schedule a job for execution.

  This function module greatly simplifies scheduling a job. You need only name an ABAP report. The function module presents screens to your user to allow them to 1) specify the variant to use; and 2) pick a single or repetitive start time for the job.

- BP_JOBVARIANT_OVERVIEW [Page 15]: Manage jobs.

  This function module offers a simplified management interface to jobs. A user can delete inactive jobs and display job logs and spool output from completed jobs.

For more information, please see the online documentation in the function module facility (transaction SE37) or Scheduling and Managing a Job: Easy Method [Page 12].

BP_CALCULATE_NEXT_JOB_STARTS: Determine Start Dates and Times for a Periodic Job

Use this function module to calculate the dates and times upon which a periodic job would be started. You can specify the period of time for the analysis.

For more information, please see the online documentation in the function module facility (transaction SE37) or Calculating Start Dates for Periodic Jobs [Page 25].

BP_CHECK_EVENTID: Check that an Event Exists

Check that the event name that you specify has been defined in the R/3 System (with transaction SM62). A return code of 0 indicates that the event is defined in the R/3 System.

For more information, please see the online documentation in the function module facility (transaction SE37).

BP_EVENT_RAISE: Trigger an Event from an ABAP Program

Trigger an event in the background processing system. This function module is for use in programs written in ABAP. Triggering an event tells the background processing system to start any background jobs that were scheduled to wait for the event.

You can trigger an event with or without an event argument, a string that more precisely identifies an event. Jobs can be scheduled to wait for an event for the combination of event and a particular event argument.

For more information, see "Triggering Events from ABAP Programs" in Using Events to Trigger Job Starts [Page 55].

Use the program SAPEVT to trigger an event from a program, script, or.bat file that runs outside the R/3 System.

For more information, please see the online documentation in the function module facility (transaction SE37) or Using Events to Trigger Job Starts [Page 55].

BP_JOB_COPY: Copy a Background Job

Copy a background job. The copy of the job includes job steps and associated attributes such as the user for authorizations and printing/archiving specifications. However, the copy has no start specifications. These you must specify yourself.

You can use the function module in two modes:

- Interactive (DIALOG = BTC_YES): the user is asked for a name for the job and must confirm or terminate the copy action. Any default name that you specify with TARGET_JOBNAME is ignored.
- Non-interactive (DIALOG = BTC_NO): The job is copied without user interaction. You must specify a new job name with TARGET_JOBNAME.

For more information, please see the online documentation in the function module facility (transaction SE37).

BP_JOB_DELETE: Delete a Background Processing Job

Delete a background job and its log (if the job has already run).

Deletion is unconditional, except that active jobs cannot be deleted and a job cannot delete itself. Aborted jobs, however, are deleted.

If it is important that aborted jobs and their logs are kept, then you may wish to specify automatic deletion with the DELANFREP parameter of JOB_OPEN. This parameter causes automatic deletion only of jobs that have been successfully completed.

For more information, please see the online documentation in the function module facility (transaction SE37) or Selecting and Deleting a Job [Page 53].

BP_JOB_GET_PREDECESSORS: List Predecessor-Jobs of a Job

Lists the immediate predecessor job of a job. Predecessor jobs are those that must be completed (optionally, successfully completed) before the affected job can be started.

The function module returns only the immediate predecessor job, the job upon whose completion this job can start. The function module does not determine whether the predecessor job itself has a predecessor.

For more information, please see the online documentation in the function module facility (transaction SE37).

BP_JOB_GET_SUCESSORS: List the Successor-Jobs of a Job

Lists the successor jobs of a job. Successor jobs are those that are eligible to start only after this job has been completed (optionally, completed successfully).

The function module may return more than one successor if successor jobs themselves have successors.

For more information, please see the online documentation in the function module facility (transaction SE37).

BP_JOB_MAINTENANCE: Job Management Functions

Starts the background processing job management function (transaction SM37). Users must be authorized for such actions as releasing jobs or accessing jobs other than their own.

For more information, please see the online documentation in the function module facility (transaction SE37) or Managing Jobs: Generating Job Lists [Page 47].

BP_JOB_SELECT: Read Jobs from Database

Returns the set of jobs (records of the form TBTCJOB) that match the selection criteria that you specify.

You can use BP_JOB_SELECT to collect a set of jobs that you wish to process further. For example, you could select jobs that match a particular naming convention and check their status with the field TBTCJOB-STATUS.

For a sample program containing BP_JOB_SELECT, please see the online documentation in the function module facility (transaction SE37) Selecting and Deleting a Job [Page 53].

BP_FIND_JOBS_WITH_PROGRAM: Read Jobs that Run a Specific Program from Database

You can use this function module as you would BP_JOB_SELECT to generate a list of jobs that can be displayed and manipulated with BP_JOBLIST_PROCESSOR. This function module selects jobs by the program that you specify. You could, for example, select the jobs that contain the program that a user has just scheduled.

For more information, please see the online documentation in the function module facility (transaction SE37). For a programming example, please see Displaying Job Status: SHOW_JOBSTATE [Page 50].

BP_JOBLIST_PROCESSOR: Allow User to Work with List of Jobs

You can use the function module BP_JOBLIST_PROCESSOR to display a list of jobs for the users of your program. The jobs are displayed in the "job overview" list format used by the job maintenance facility of the background processing system.

The list of jobs must be provided to BP_JOBLIST_PROCESSOR in an internal table generated with BP_JOB_SELECT.

You can display the list of jobs in three modes:

- Display mode: The user can view job attributes and such job products as spool requests and job logs. He or she cannot, however, change jobs or perform such operations as releasing a job to run.
- Edit mode: The user can perform all maintenance operations for which he or she is authorized.
- Select mode: The user is requested to select a job from the list. When the selection is made, the list is closed and the TBTCJOB structure of the selected job is returned to your program.

Authorizations

The authorization tests for background processing are in effect for processing of job lists. These authorizations are described in the *User and Authorizations Guide* and in Basis Customizing.

For more information, please see the online documentation in the function module facility (transaction SE37).

SHOW_JOBSTATE: Check Status of a Job

Returns one of the six possible statuses for a job that is identified by its name and job number.

For sample code, please see Displaying Job Status: SHOW_JOBSTATE [Page 50].

BP_JOBLOG_READ: Read a Job Log for Processing

Read the contents of a job processing log into an internal table for further processing. You could, for example, combine the logs of related jobs into a single log. Display sample code

For sample code, please see the online documentation in the function module facility (transaction SE37) or Copying a Job Log into an Internal Table [Page 46].

BP_JOBLOG_SHOW: Display a Job Processing Log

Display the contents of a job processing log.

A job log contains messages from the background processing system on the processing of a job, together with any error messages from the ABAP programs executed in the job and/or output from external programs. Display sample code

For sample code, please see the online documentation in the function module facility (transaction SE37) Job Outcomes: Displaying Job Logs [Page 44].

BP_START_DATE_EDITOR: Display/Request Start Specifications

Use this function module to let users specify the start time for a job themselves. The function module displays the standard scheduling screens for background processing jobs to users.

For more information, please see the online documentation in the function module facility (transaction SE37) or Getting Job-Start Specifications from Users [Page 22].

BP_JOB_READ: Retrieve Job Specifications

Use this function module to read the job header table (TBTCO, specifications such as the start time that apply globally to a job) and the job step table (TBTCSTEP, a record for each step in a job, including such information as the program to run, spool request generated, and so on).

You'll need the job name and job number to read these tables.

What the function module returns is controlled by the import parameter JOB_READ_OPCODE. This parameter can take on the following values (defined in LBTCHDEF):

- BTC_READ_JOBHEAD_ONLY: Read only the job header data (TBTCO)
- BTC_READ_ALL_JOBDATA: Read all job data (TBTCO, TBTCSTEP).

There is currently no programming example for this function module.

SHOW_JOBSTATE: Display Job Status

Use this function module to return the status of a job in the background processing system. The statuses are those that you can display in the management transaction for background processing (transaction SM37).

For a programming example, please see Displaying Job Status: SHOW_JOBSTATE [Page 50].

Parallel-Processing Function Modules

Three function modules can be used in association with the ABAP CALL FUNCTION STARTING NEW TASK IN DESTINATION GROUP, WAIT, and RECEIVE keywords to implement parallel processing in programs run interactively or in the background processing system.

For a list of these function modules and more information on their use, please see Implementing Parallel Processing [Page 65].

Data Transfer

Purpose

During data transfer, data is transferred from an external system into the SAP R/3 System. You can use data transfer when you:

- Transfer data from an external system into an R/3 System as it is installed.
- Transfer data regularly from an external system into an R/3 System. Example: If data for some departments in your company is input using a system other than the R/3 System, you can still integrate this data in the R/3 System. To do this, you export the data from the external system and use a data transfer method to import it into the R/3 System.

Implementation considerations

Before creating your own data transfer program, you should use the Data Transfer Workbench [Ext.] to find the data transfer programs that are delivered by SAP.

Integration

SAP applications support the data transfer of numerous SAP business objects. The data transfer program specifies the data format definition that is necessary to import the data into the R/3 System. Adapt your conversion program for exporting the data from the external system to this definition.

Once the data has been exported, you can import it into your system using a generated data transfer program.

Features

- Data transfer from other, external systems
- Generation of data transfer programs
- Generation of function modules that can be used as an interface to the R/3 System

Data Transfer Methods

You can use the following methods to transfer data:

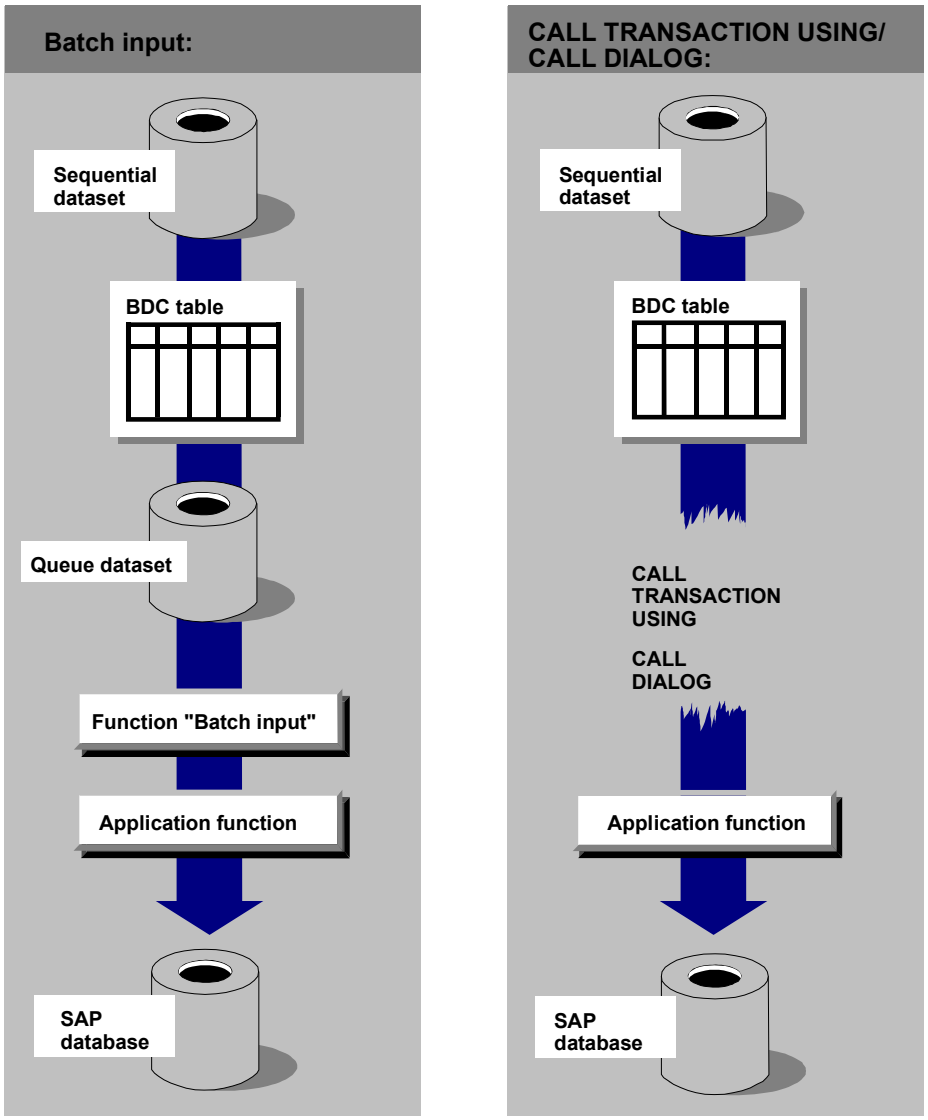
- **Direct input**: With direct input [Page 148], the SAP function modules execute the consistency checks. However with batch input, these consistency checks are executed with the help of the screens. This means that direct input has considerable performance advantages.
- **`CALL TRANSACTION`:** Data consistency check with the help of screen logic.
- **Batch input with batch input sessions:** Data consistency check with the help of screen logic.

Difference between Batch Input and CALL TRANSACTION

If the direct input cannot be used for your task, this makes creating a data transfer program easier since the underlying transactions ensure that the data consistency checks are executed.

In the case of an error during the data transfer (if data records are inconsistent, for example), you can restart the transfer at the point in the program where the error occurred.

Data Transfer Overview



Batch input methods

With the batch input method, an ABAP program reads the external data that is to be entered in the R/3 System and stores the data in a “batch input session”. The session records the actions that are required to transfer data into the system using normal SAP transactions.

When the program has generated the session, you can run the session to execute the SAP transactions in it. You can explicitly start and monitor a session with the batch input management function (by choosing *System* → *Services* → *Batch input*), or have the session run in the background processing system.

CALL TRANSACTION methods

Data Transfer Methods

In the second method, your program uses the ABAP statement CALL TRANSACTION USING to run an SAP transaction. External data does not have to be deposited in a session for later processing. Instead, the entire batch input process takes place inline in your program.

The information in Choosing Data Transfer Methods [Page 126] will help you decide which is the best data transfer method.

Data Transfer: Overview of Batch Input

Prerequisites

Before beginning the initial data transfer, you should answer the following questions:

- Which business objects are to be transferred?
  The business objects to be transferred depend on the business applications that you will be using. If you are using sales and distribution processing, for example, you must transfer the material masters as well as the sales documents from your legacy system.
- How are the business objects to be transferred?
  The data can be transferred either manually or automatically, using an SAP data transfer program or using your own transfer program.

Process flow

During the initial data transfer, data from the external system is converted into a sequential data transfer file and then transferred into the R/3 System using an SAP data transfer program. The data transfer file is the prerequisite for successfully transferring data as it contains the data in a converted format that is suitable for the R/3 System.

1. Check to see if an SAP data transfer program (direct input, batch input or CALL TRANSACTION) exists for this data using the Data Transfer Workbench [Ext.]. Refer to the notes for this transfer program.

   If no SAP data transfer program exists, proceed as follows:
2. Determine the SAP transactions that a user would use to enter data records.
3. Record [Page 111] these transactions using the batch input recorder. Ensure that you have filled all of the relevant fields with data.
4. Use this data to generate [Page 115] a data transfer program.
5. Display the Data Transfer Workbench [Ext.] and create your own data transfer object.
6. Now follow the steps for transferring data using the Data Transfer Workbench.

The Transaction Recorder

Use

You can use the transaction recorder to record a series of transactions and their screens.

Features

You can use the recording to create

- Data transfer programs that use batch input or CALL TRANSACTION
- Batch input sessions
- Test data
- Function modules.

The recording can be executed several times. The screens are run in exactly the same way as they were run during the recording.

You can edit recordings that you have already made using an editor.

Activities

1. To display the batch input initial screen, choose *System* → *Services* → *Batch input*→ *Edit.*
2. Choose *Recording.* The system now displays the initial screen of the batch input recorder.
3. Make a recording of the relevant transactions.
4. Generate one or several of the objects named above.

Special features

- `F1`-, `F4`- and self-programmed `F1`- and `F4` help (**`PROCESS ON HELP-REQUEST`**,**`PROCESS ON VALUE-REQUEST`**) are not recorded. The same applies to all commands in the *System* and *Help* menus.
- Error and warning dialogs are not recorded. This means that only the OK code field and the field contents that lead to successful further processing in the current screen.
- "**`COMMIT WORK`**" in a transaction flow indicates the successful end of the transaction as regards batch input. The recording also ends successfully.
- "**`LEAVE TO TRANSACTION`**" in the course of a transaction indicates that it is not suitable for batch input. The recording is terminated.
- In ScreenPainter screens, movements in the scrollbar are not recorded. Use the function keys `F21-F24` for positioning.

Recording Transactions

The recording forms the basis of generating data transfer programs, sessions, test data and function modules.

Procedure

1. Display the initial screen of the batch input recorder.
2. Assign a name to your recording.
3. Choose *Create*.
4. On the subsequent dialog box, enter the transaction code that you want to record and choose *Continue*.
   The system displays this transaction.
5. Execute the transaction in the usual way.
6. When you have finished processing the transaction, the system displays an overview of the transaction and your input.
   Choose *Get transaction* if no errors occurred while the transaction was being recorded.
   If you do not want to keep the last recording that you made, go to the next step.
7. Choose *Next transac.* If you want to record an additional transaction. Then continue from point 4.
8. Save your recording when you have finished.

Recording

Once you have recorded the transaction, you can process it again later.

Procedure

1. Display the initial screen of the batch input recorder (Transaction *SHDB*).
2. Choose *Overview*.
   The system displays an overview of all recordings.
3. Position the cursor on the relevant recording and choose *Execute*.
4. Choose one of the following processing modes:

   A: Display all screens.

   E: Display errors only. In the case of an error, the system displays the screen on which the error occurred. Once this error has been corrected, the system continues to process the recording until the next error occurs.

   N: No display. The recording is not processed visibly.
5. Choose the update mode:

   A: Asynchronous update

   S: Synchronous update

   L: Local update
6. You begin to process the recording when you choose *Enter* to exit the dialog box. When the recording is complete, the system displays a log that lists the name of the transaction, the system field **SY-SUBRC** and the system messages that were output.

Using the Recording Editor

The recording editor contains functions that you can use to edit your recordings.

Procedure

1. Display the initial screen of the batch input recorder (Transaction *SHDB*).
2. Choose *Overview*.
   The system displays an overview of all recordings.
3. Position the cursor on the relevant recording and choose *Change*.
4. The following functions are available on the overview that the system displays:
   - Delete transaction from the recording: This deletes the selected transaction.
   - Add a new transaction to the recording: The transaction is added at the end of the recording.
   - Editing: You can edit the current recording.

     If you choose *Editing,* proceed as follows:
5. The system displays an editor where you can add and delete individual lines. You can also change the contents of these lines.
6. If these editor functions are insufficient for your requirements, you can choose *Export* to download the recording onto your presentation host and use a PC editor to edit it there.
7. Choose *Import* to import this file back into the R/3 System. Ensure that the file is still in the correct format.
8. Choose *Check* to ensure that the edited version of the recording is still syntactically correct.
9. Save your changes to the recording when you have finished editing it.

Generating Batch Input Sessions From the Recording

The data from the recording is transferred into batch input sessions that you can process for test purposes using the usual mechanisms.

Prerequisites

Before you can generate a batch input session, you must record the transactions through which the data is to enter the R/3 System. You use the batch input recorder [Page 110] to do this.

Procedure

1. Display the initial screen of the batch input recorder (Transaction *SHDB*).
2. Choose *Overview*.
   The system displays an overview of all recordings.
3. Position the cursor on the relevant recording and then choose *Generate session*.
4. Enter a session name, a user with whose authorizations the session is to be processed, the identification whether the session is to be deleted once it has been processed and the processing date.
5. Choose *Continue* to exit the dialog box.
   You have generated a batch input session that uses the same data for the input fields that you entered when you created the recording. You can now process [Page 141] this as usual.

Generating Data Transfer Programs

Prerequisites

Before you can generate a data transfer program, you must record the transactions using which the data is imported into the R/3 System. You use the batch input recorder [Page 110] to do this.

Procedure

1. Display the initial screen of the batch input recorder (Transaction SHDB).
2. Choose *Overview*.
   The system displays an overview of all recordings.
3. Position the cursor on the relevant recording and choose *Create program*.
4. On the following screen, specify a program name.
5. You can also choose how the maintained field contents of the recorded screens are to be filled:
   - Transfer the values that were used during the recording. If you require a flexible data transfer, you must modify this program.
   - Set the parameters of the input values to be maintained and import these values from a data file. To set the parameters, the system creates a data structure and imports the data records from an external file into this data structure. The program assumes that the external file has been adapted to this data structure.
6. If you have decided to set parameters for the input values to be maintained, it is useful if you create a test file when you generate the program. To do this, flag the checkbox and enter a name for the test file. For more information, see creating a test file [Page 118].
7. Choose *Continue*.
8. The system displays the attribute screen of the program editor. Choose the relevant attributes here and save the program.

Result

You have now generated a data transfer program that you can use to import data into the R/3 System. The program can execute the data transfer using batch input or `CALL TRANSACTION`.

Generating Function Modules

Prerequisites

Before you can generate a data transfer program, you must record the transactions using which the data is imported into the R/3 System. You use the batch input recorder [Page 110] to do this.

Procedure

1. Display the initial screen of the batch input recorder (Transaction SHDB).
2. Choose *Overview*.
   The system displays an overview of all recordings.
3. Position the cursor on the relevant recording and choose *Create function module*.
4. On the subsequent dialog box, enter a function module name, a function group and a short text for the function module. Exit the dialog box by choosing *Continue*.
   The system automatically creates the function module.

Result

You have now generated a function module that you can use as an interface for your R/3 System. As well as information relevant for the data transfer, the function module's import interface has a parameter for each input field of the transaction recorded.

Using Function Modules

Prerequisites

The function module was generated from a recording made using the batch input recorder.

Procedure

1. Cal the function module.
2. Supply the generic interface of the function module:

   CTU: Flag whether the data is to be transferred using batch input method **CALL TRANSACTION USING**. The system generates a batch input session if this flag is not set.

   MODE: Processing mode:

A	Display all
E	Display only errors
N	No display

   UPDATE: Update mode:

S	Synchronous
A	Asynchronous
L	Local update

   GROUP: (If CTU is already specified): Name of the batch input session to be generated

   USER: (If CTU is already specified): User with whose authorizations the session is to be processed

   KEEP: Specifies whether this session is to be deleted once it has been processed

   HOLDDATE: Specifies the earliest processing date for the error session

   NODATA: Defines the NODATA character [Page 125]

3. Supply the function module's special interface.
   For each input field that was filled when you recorded the transactions, the system creates an import parameter. The recorded value is used as the default value for this import parameter.

Creating Test Files

To test the data transfer program [Page 115] that you have created, you can create a data record in a sequential file. This data record contains all the field contents from the recording that are relevant to the data transfer in the format required by the data transfer program. It is therefore useful if you align the format of your conversion program [Page 120] data file with the format of the test file.

Prerequisites

Before you can generate a data transfer program, you must record the transactions using which the data is imported into the R/3 System. You use the batch input recorder [Page 110] to do this.

Procedure

1. Display the initial screen of the batch input recorder (Transaction SHDB).
2. Choose *Overview*.
   The system displays an overview of all recordings.
3. Position the cursor on the relevant recording and choose *Create test data*.
4. Enter a test file and exit the dialog box by choosing *Continue*.
   You have now created a test file.

If the test file you have specified already exists, the system appends the new data record.

If you do not specify the path, the system archives the test file in the working directory of the current application server.

Executing the Data Transfer

Purpose

You generally use the Data Transfer Workbench to execute the data transfer. The following section describes how you transfer data directly using the batch input method.

Prerequisites

You require a data transfer program. This may be an SAP data transfer program, or you can create your own program.

Process flow

1. Provide the data to be imported in a data file [Page 120]. Ensure that the data is in the correct format.
2. If you are using a generated data transfer program [Page 115], you can choose a data transfer method.
   If you are only dealing with one data record, you can import this directly using a generated function module [Page 116].
3. Execute the data transfer program [Page 128].
4. Analyze the program and correct any errors that occur.

Writing Data Conversion Programs

The data conversion program is responsible for the following tasks:

- Converting the data that is to be transferred into the R/3 System as required by the SAP data structure or transactions that you are using.

  If you are using an SAP batch input standard program, you must generate the data structure from the SAP standard data structure (see generating an SAP data structure [Page 122]).

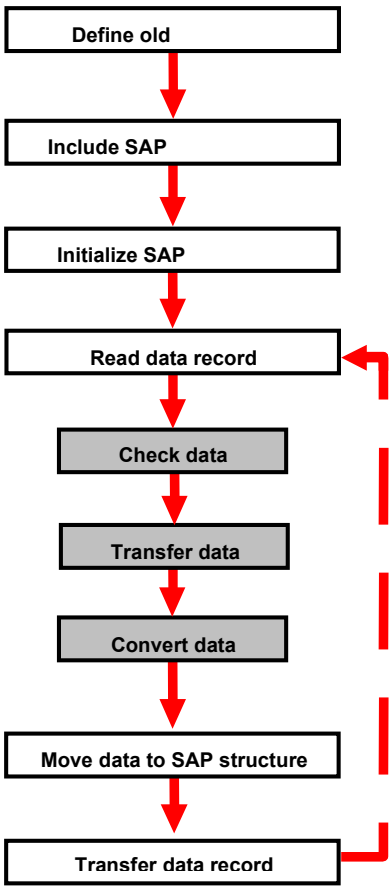
  If you develop your own batch input program, the data structure is determined by the R/3 System when the program is generated. Generate a test file from the recording and align the format of your conversion program with the format of the test file.

  A conversion may be necessary for data type and length [Page 124] data type and length. The data type required by all standard SAP batch input programs is C, character data. You can find the required field lengths either in your analysis of the data declaration structure of the generated batch input program or in the data structures that you generate.

- The data is exported in SAP format to a sequential file. The batch input program in the R/3 System reads the data in from this file.

Process flow

The tasks involved in writing a data transfer program are shown in the diagram and list below.



1. Analyze the structure of your existing data and specify the conversions that are required to fill the SAP data structures.
2. If necessary, generate [Page 122] the SAP data structure in code form and insert it into your program.
3. This is only possible if the SAP data structure is an ABAP Dictionary object (structure, table).
4. Fill [Page 125] the structure with data. If you do not want to assign a value to a field in the structure, use the NODATA character. Execute any conversions [Page 124] conversions and error checks that may be required.
5. Write the sequential file that is typically required for making the data available to the data transfer program in the R/3 System.

For SAP data transfer standard programs, all data must have the character format.

Generating an SAP Data Structure for the Conversion Program

You can use the ABAP Dictionary to generate data structures for ABAP Dictionary objects (tables, views and structures) in any of the following programming languages:

- Cobol
- PL/1
- C
- RPG

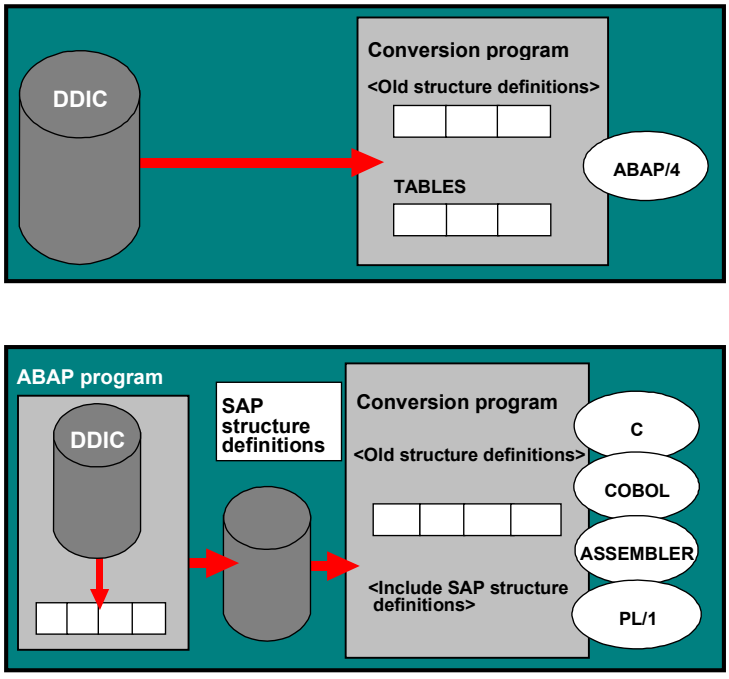
You can then incorporate these data structures in your conversion program.

For most of the SAP data transfer standard programs, SAP has also provided special data structure identifiers. With such an identifier, you can generate a listing of all of the table fields that are required by the data transfer program. You manage these programs in the Data Transfer Workbench [Ext.]. If you have generated your own program to transfer data from an external file, the SAP data structure definition that you require is stored in the data declaration section of the program.

Conversion Programs in ABAP

If your conversion program is written in ABAP, you can use the information from the ABAP Dictionary directly to record the data structure of your tables.

The graphic below shows the structures of conversion programs in ABAP and as external programs written in other languages.



Including Table Structures in Conversion Programs: ABAP and Other Languages

Procedure

To generate a data structure, proceed as follows:

1. Choose *Tools → ABAP Workbench* and then *ABAP Dictionary*.
2. In the *ABAP* Dictionary, select *Environment → Generate table description.*

   With this function, you can generate the structures of ABAP Dictionary objects in programming code. You can then add this code to your conversion program.
3. Specify the programming language in which the structure should be generated and identify the tables to be included.

   If you wish to use a special structure identifier for a standard SAP batch input standard program, then enter the identifier in the *Key in TSRCG* field. Example: The identifier **AM-ANLA** generates the data structure required for data transfers for the asset management application.
4. The system displays the data structure in list form, as in this example in PL/1 from the asset management application (AM-ANLA):

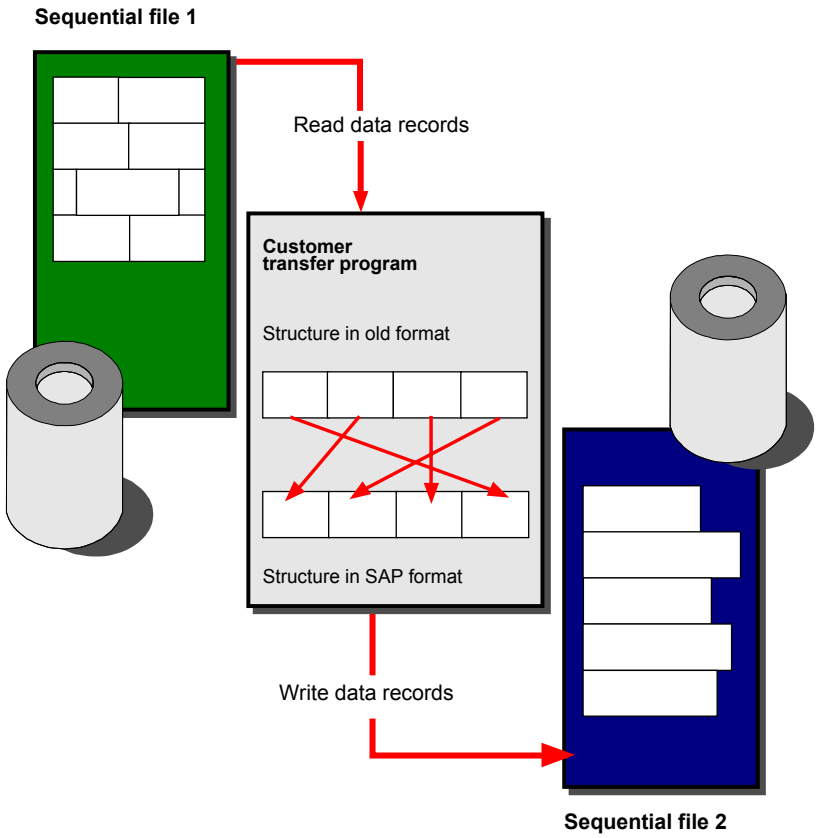
```
*******************************************************************
*
   *    MEMBER GENERATED FROM SAP DATA DICTIONARY        *
   *         T A B L E  BALTD                *
   *    DATE: 08.12.1995   TIME: 17:47:16          *
   *       PLEASE DO NOT CHANGE MANUALLY          *

*******************************************************************
*
   *
    01 BALTD.
   *
   * Client (Old Assets Data Takeover AM)
      05  MANDT          PIC X(3)
                     VALUE SPACE.
   * Company code
      05  BUKRS          PIC X(4)
                     VALUE SPACE.
   * Asset class
      05  ANLKL          PIC X(8)
                     VALUE SPACE.
```

   Choose *System → List → Save → File* to download the data structure to your workstation or PC.

Data Conversion

Once you have determined the SAP data structure, you should:

- Determine which fields can be transferred directly from your existing data. There is a direct match between an existing data field and the corresponding SAP data field.
- Check which fields must be converted to adapt the existing data to the requirements of the R/3 System.



Converting data from the old format to SAP format

Your program must format legacy data just as an online user would when typing them in. In particular:

- The data must be character format.
- The data itself must not be longer than its target SAP field.
- If the data is shorter than the target field, you must left-justify it within the SAP field (pad it with blank characters at the right end).
- If the transaction for a field is to keep its initial value, use the NODATA character [Page 125].

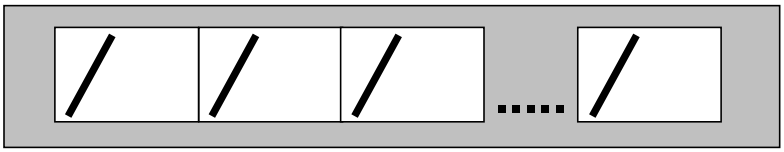
Filling SAP Data Structures

Standard SAP data transfer programs require that every field in a data structure contains either:

- a value; or
- a special NODATA marker, which indicates that the no batch input data is required for the field.

If you are writing your own conversion program, you should therefore initialize all of the fields in your batch input data structure with the NODATA character. The NODATA character must occupy the first position in the field, as shown in the figure below.

By default, the NODATA character is the forward slash. To initialize a field to NODATA, you must write this character as the first character in the field value.



If a batch input program finds NODATA in a field, then the program allows the field to default to its standard value in the SAP transaction that contains the field.

Setting the NODATA character: You can freely select another character as the NODATA character by:

- Specifying this in the selection screen for a generated data transfer program
- Defining the new character in the data transfer program that reads the data in the BGR00-NODATA field:

```
Data: <Name> like bgr00.
<Name>-NODATA = '<Character>'.
```

Selecting a Data Transfer Method

When you transfer data in ABAP, you have three options to submit the data for the data transfer. Only the first two methods can be recommended without reservation. The third method, by way of CALL DIALOG, is outmoded. CALL DIALOG is less comfortable than the other methods. You should use it only if you must.

- Use the **CALL TRANSACTION USING** statement

  Summary: With **CALL TRANSACTION USING,** the system processes the data more quickly than with batch input sessions. Unlike batch input sessions, **CALL TRANSACTION USING** does not automatically support interactive correction or logging functions.

  Your program prepares the data and then calls the corresponding transaction that is then processed immediately.

  The most important features of **CALL TRANSACTION USING** are:

  - Synchronous processing
  - Transfer of data from an individual transaction each time the statement **CALL TRANSACTION USING** is called
  - You can update the database both synchronously and asynchronously

    The program specifies the update type
  - Separate LUW (logical units of work) for the transaction

    The system executes a database commit immediately before and after the **CALL TRANSACTION USING** statement
  - No batch input processing log

- Create a session on the batch input queue.

  Summary: Offers management of sessions, support for playing back and correcting sessions that contain errors, and detailed logging.

  Your program prepares the data and stores it in a batch input session. A session is a collection of transaction data for one or more transactions. Batch input sessions are maintained by the system in the batch input queue. You can process batch input sessions in the background processing system.

  Your program must open a session in the queue before transferring data to it, and must close it again afterwards. All of these operations are performed by making function module calls from the ABAP program.

  The most important aspects of the session interface are:

  - Asynchronous processing
  - Transfers data for multiple transactions
  - Synchronous database update

    During processing, no transaction is started until the previous transaction has been written to the database.

- A batch input processing log is generated for each session
- Sessions cannot be generated in parallel

  The batch input program must not open a session until it has closed the preceding session.

- Use the CALL DIALOG statement

  Summary: Not recommended if you can enter data by way of sessions or CALL TRANSACTION USING.

  Your program prepares data for a sequence of dialog screens, and calls a dialog module for immediate processing.

  The most important aspects of the CALL DIALOG interface are:

  - Synchronous processing
  - Transfers data for a sequence of dialog screens
  - No separate database update for the dialog

    A database update occurs only when the calling program executes a commit operation.
  - Shares LUW with calling program
  - No batch input processing log is generated

Executing Data Transfer Programs

Procedure

If you are using an SAP data transfer program, follow the procedure specified in the program documentation.

If you are using a generated data transfer program, proceed as follows:

1. Start the data transfer program.
2. Decide which batch input method you want to use for the data transfer.

   a) CALL TRANSACTION USING:

   You must specify the:

   – *Processing mode*: You use this parameter to specify whether processing should take place in the background or in dialog mode.
   Possible values are:

A	Display all
E	Display only errors
N	No display

   – *Update mode*: This parameter determines how the data is to be updated:
   Possible values are:

S	Synchronous
A	Asynchronous
L	Local update

   – *Error session*: Here you have the option to specify a session name for a batch input session in which data is to be written in the case of an error. You can use this to identify incorrect data records after the batch input program has run and to import the records into the R/3 System once you have corrected them.

   If you are creating an error session, you must also specify:

   – *User*: Specify the user with whose authorizations the sessions are processed.

   – *Keep session*: This specifies whether or not the session should be deleted once it has been processed.

   – *Lock date*: Specify the processing date for the error session.

   b) Generate session:

   – *Session name*: Specify a name for the batch input session to be generated.

   – *User*: Specify the user with whose authorizations the sessions are processed.

   – *Keep session*: This specifies whether or not the session should be deleted once it has been processed.

   – *Lock date*: Specify the processing date for the error session.

3. Specify a character that is to be used as the NODATA character.

4. Specify the path of the data file from which the data is to be imported into the R/3 System.
5. Execute the program.
6. If you have generated a session, or if errors occurred in **`CALL TRANSACTION USING`** mode, you must now edit the generated sessions. You can find information on this in *BC - System services* in batch input sessions [Ext.].

Batch Input Authorizations

You do not need special authorization - other than the ABAP run time authorization (authorization object **S_PROGRAM** - to run a program that creates batch input. Any user can create batch input sessions.

Starting processing for a session once it is in the queue is another matter, however. You can find more information on this in batch input sessions [Ext.].

Additional Information

In this section you can find detailed technical information. You need this information if you want to create your own batch input program without generating it, or you want to change a program that has been generated.

Using CALL TRANSACTION USING for Data Transfer

Processing batch input data with CALL TRANSACTION USING is the faster of the two recommended data transfer methods. In this method, legacy data is processed inline in your data transfer program.

For more information, see choosing a data transfer method [Page 126].

Syntax:

```
CALL TRANSACTION  <tcode>
       USING  <bdc_tab>
       MODE   <mode>
       UPDATE  <update>
```

<tcode>: Transaction code

<bdc_tab>: Internal table of structure BDCDATA [Page 143].

<mode>: Display mode:

A	Display all
E	Display errors only
N	No display

<update>: Update mode:

S	Synchronous
A	Asynchronous
L	Local update

A program that uses **CALL TRANSACTION USING** to process legacy data should execute the following steps:

1. Prepare a BDCDATA [Page 143] structure for the transaction that you wish to run.
2. With a CALL TRANSACTION USING statement, call the transaction and prepare the BDCDATA structure. For example:

```
CALL TRANSACTION 'TFCA' USING BDCDATA
        MODE 'A'
        UPDATE 'S'.
        MESSAGES INTO MESSTAB.

IF SY-SUBRC <> 0.
  <Error_handling>.
ENDIF.
```

The MODE Parameter

You can use the **MODE parameter** to specify whether data transfer processing should be displayed as it happens. You can choose between three modes:

A Display all. All screens and the data that goes in them appear when you run your program.

N No display. All screens are processed invisibly, regardless of whether there are errors or not. Control returns to your program as soon as transaction processing is finished.

E Display errors only. The transaction goes into display mode as soon as an error in one of the screens is detected. You can then correct the error.

The display modes are the same as those that are available for processing batch input sessions.

The UPDATE Parameter

You use the **UPDATE parameter** to specify how updates produced by a transaction should be processed. You can select between these modes:

A Asynchronous updating. In this mode, the called transaction does not wait for any updates it produces to be completed. It simply passes the updates to the SAP update service. Asynchronous processing therefore usually results in faster execution of your data transfer program.

> Asynchronous processing is NOT recommended for processing any larger amount of data. This is because the called transaction receives no completion message from the update module in asynchronous updating. The calling data transfer program, in turn, cannot determine whether a called transaction ended with a successful update of the database or not.
>
> If you use asynchronous updating, then you will need to use the update management facility (Transaction SM12) to check whether updates have been terminated abnormally during session processing. Error analysis and recovery is less convenient than with synchronous updating.

S Synchronous updating. In this mode, the called transaction waits for any updates that it produces to be completed. Execution is slower than with asynchronous updating because called transactions wait for updating to be completed. However, the called transaction is able to return any update error message that occurs to your program. It is much easier for you to analyze and recover from errors.

L Local updating. If you update data locally, the update of the database will not be processed in a separate process, but in the process of the calling program. (See the ABAP keyword documentation on **`SET UPDATE TASK LOCAL`** for more information.)

The MESSAGES Parameter

The **`MESSAGES`** specification indicates that all system messages issued during a **`CALL TRANSACTION USING`** are written into the internal table **`<MESSTAB>`**. The internal table must have the structure **`BDCMSGCOLL`**.

You can record the messages issued by Transaction TFCA in table **`MESSTAB`** with the following coding:

(This example uses a flight connection that does not exist to trigger an error in the transaction.)

```
DATA: BEGIN OF BDCDATA OCCURS 100.
    INCLUDE STRUCTURE BDCDATA.
DATA: END OF BDCDATA.
DATA: BEGIN OF MESSTAB OCCURS 10.
    INCLUDE STRUCTURE BDCMSGCOLL.
DATA: END OF MESSTAB.
BDCDATA-PROGRAM = 'SAPMTFCA'.
BDCDATA-DYNPRO = '0100'.
BDCDATA-DYNBEGIN = 'X'.
APPEND BDCDATA.
CLEAR BDCDATA.
BDCDATA-FNAM = 'SFLIGHT-CARRID'.
BDCDATA-FVAL = 'XX'.
APPEND BDCDATA.
BDCDATA-FNAM = 'SFLIGHT-CONNID'.
BDCDATA-FVAL = '0400'.
APPEND BDCDATA.
CALL TRANSACTION 'TFCA' USING BDCDATA MODE 'N'
         MESSAGES INTO MESSTAB.
LOOP AT MESSTAB.
  WRITE: / MESSTAB-TCODE,
     MESSTAB-DYNAME,
     MESSTAB-DYNUMB,
     MESSTAB-MSGTYP,
     MESSTAB-MSGSPRA,
     MESSTAB-MSGID,
     MESSTAB-MSGNR.
ENDLOOP.
```

The following figures show the return codes from **CALL TRANSACTION USING** and the system fields that contain message information from the called transaction. As the return code chart shows, return codes above 1000 are reserved for data transfer. If you use the **MESSAGES INTO <table>** option, then you do not need to query the system fields shown below; their contents are automatically written into the message table. You can loop over the message table to write out any messages that were entered into it.

Return codes:

Value	Explanation
0	Successful
<=1000	Error in dialog program
> 1000	Batch input error

System fields:

Name:	Explanation:
SY-MSGID	Message-ID

SY-MSGTY	Message type (E,I,W,S,A,X)
SY-MSGNO	Message number
SY-MSGV1	Message variable 1
SY-MSGV2	Message variable 2
SY-MSGV3	Message variable 3
SY-MSGV4	Message variable 4

Error Analysis and Restart Capability

Unlike batch input methods using sessions, CALL TRANSACTION USING processing does not provide any special handling for incorrect transactions. There is no restart capability for transactions that contain errors or produce update failures.

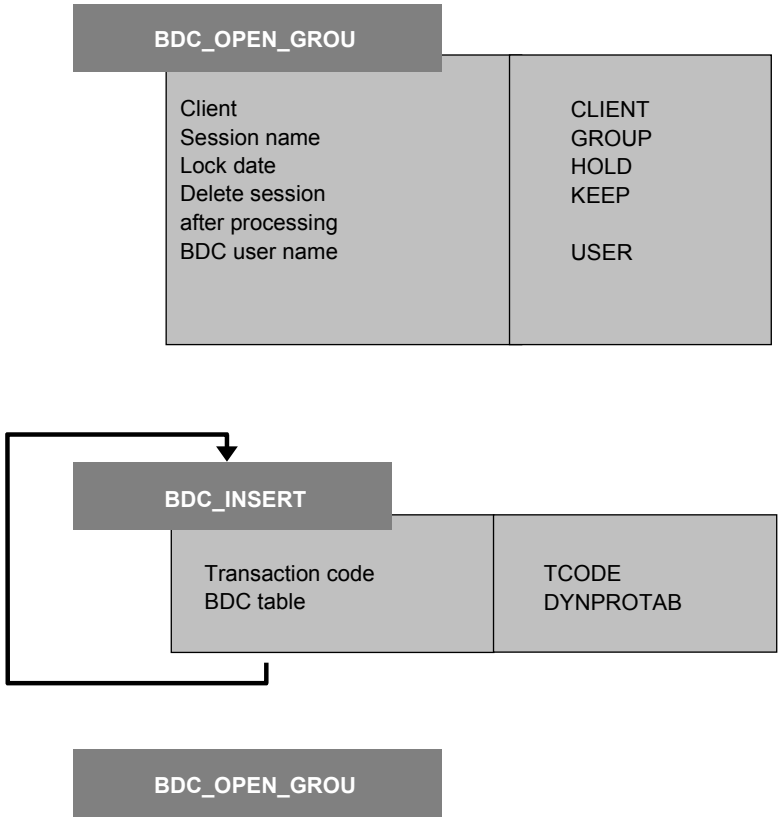
You can handle incorrect transactions by using update mode S (synchronous updating) and checking the return code from CALL TRANSACTION USING. If the return code is anything other than 0, then you should do the following:

- write out or save the message table
- use the BDCDATA table that you generated for the CALL TRANSACTION USING to generate a batch input session for the faulty transaction. You can then analyze the faulty transaction and correct the error using the tools provided in the batch input management facility.

Creating Batch Input Sessions

One of the two recommended ways to process batch input data is to store the data in a batch input session. You can then run this session in the R/3 System to enter the batch input data into the system.

Process flow



If you have generated a batch input program, the following function modules are automatically assigned the correct values.

If you want to write your own program, proceed as follows:

1. Generate the batch input session using function module BDC OPEN GROUP [Page 137].
2. The proceed as follows for each transaction that the session contains:
   a. In the BDCDATA structure [Page 143], enter the value for all screens and fields that must be processed in the transaction.
   b. Use BDC INSERT [Page 139] to transfer the transaction and the BDCDATA structure to the session.
3. Close the batch input session with BDC CLOSE GROUP [Page 140]

Start to process the generated session [Page 141].

Creating a Session with BDC_OPEN_GROUP

Use the BDC_OPEN_GROUP function module to create a new session. Once you have created a session, then you can insert batch input data into it with BDC_INSERT.

You cannot re-open a session that already exists and has been closed. If you call BDC_OPEN_GROUP with the name of an existing session, then an additional session with the same name is created.

A batch input program may have only one session open at a time. Before opening a session, make sure that any sessions that the program closes any sessions that it previously had opened.

```
CALL  FUNCTION  'BDC_OPEN_GROUP'

EXPORTING
   CLIENT       =   <client>
   GROUP        =   <session name>
   HOLDDATE     =   <lock date>
   KEEP         =   <deletion indicator>
   USER         =   <batch user name>

EXCEPTIONS RUNNING
   QUEUE_ERROR      =   1
   CLIENT_INVALID   =   2
   GROUP_INVALID    =   3
         .
         .
         .
```

BDC_OPEN_GROUP takes the following EXPORTING parameters:

- *CLIENT*

  Client in which the session is to be processed.

  Default: If you don't provide a value for this parameter, the default is the client under which the batch input program runs when the session is created.

- *GROUP*

  Name of the session that is to be created. May be up to 12 characters long.

  Default: None. You must specify a session name.

- *HOLDDATE*

  Lock date. The session is locked and may not be processed until **after** the date that you specify. Only a system administrator with the **LOCK** authorization for the authorization object *Batch Input Authorizations* can unlock and run a session before this date.

  Format: YYYYMMDD (8 digits).

Default: No lock date, session can be processed immediately. A lock date is optional.

- *KEEP*

  Retain session after successful processing. Set this option to the value **x** to have a session kept after it has been successfully processed. A session that is kept remains in the input/output queue until an administrator deletes it.

  Sessions that contain errors in transactions are kept even if *KEEP* is not set.

  Default: If not set, then sessions that are successfully processed are deleted. Only the batch input log is kept.

- *USER*

  Authorizations user for background processing. This is the user name that is used for checking authorizations if a session is started in background processing. The user must be authorized for all of the transactions and functions that are to be executed in a session. Otherwise, transactions will be terminated with “no authorization” errors.

  The user can be of type dialog or background. Dialog users are normal interactive users in the R/3 System. Background users are user master records that are specially defined for providing authorizations for background processing jobs.

Adding Data to a Session: BDC_INSERT

Use the BDC_INSERT function module to add a transaction to a batch input session. You specify the transaction that is to be started in the call to BDC_INSERT. You must provide a BDCDATA structure that contains all of the data required to process the transaction completely.

```
CALL  FUNCTION  'BDC_INSERT'
EXPORTING       TCODE       =  <transaction code>
TABLES       DYNPROTAB      =  <bdc_table>

EXCEPTIONS
   INTERNAL_ERROR           =  1
        NOT_OPEN            =  2
      QUEUE_ERROR           =  3
      TCODE_INVALID         =  4
```

BDC_INSERT takes the following parameters:

- TCODE

  The code of the transaction that is to be run.

- POST_LOCAL

  Parameter to update data locally. If POST_LOCAL = ‘X’, data will be updated locally.

  (refer to the keyword documentation of SET UPDATE TASK LOCAL for more information)

- DYNPROTAB

  The BDCDATA structure [Page 143] that contains the data that is to be processed by the transaction.

  DYNPROTAB is a table parameter in the function module.

Closing a Session: BDC_CLOSE_GROUP

Use the BDC_CLOSE_GROUP function module to close a session after you have inserted all of your batch input data into it. Once a session is closed, it can be processed.

Function Module BDC_CLOSE_GROUP

Exception parameters

Parameter	Function
NOT_OPEN	Client
QUEUE_ERROR	Internal use

BDC_CLOSE_GROUP needs no parameters. It automatically closes the session that is currently open in your program.

You must close a session before you can open another session from the same program.

You cannot re-open a session once it has been closed. A new call to BDC_OPEN_GROUP with the same session name creates a new session with the same name.

Processing Batch Input Sessions

When you create a batch input session, it remains in the batch input queue until it is explicitly started. Session processing can be started in two ways:

- An on-line user can start the session using the batch input menu options. (To access the batch input options, choose *System* → *Services* → *Batch Input*.)
- You can submit the background job RSBDCSUB to start a session in background processing. If several sessions have the same name, RSBDCSUB starts them all.

  It's possible to coordinate the generation and execution of a session in the background processing system.

  You can, for example, schedule both the batch input program and RSBDCSUB in the background. If you designate the batch input job as the predecessor for RSBDCSUB, then RSBDCSUB will be started automatically when the batch input job successfully completes.

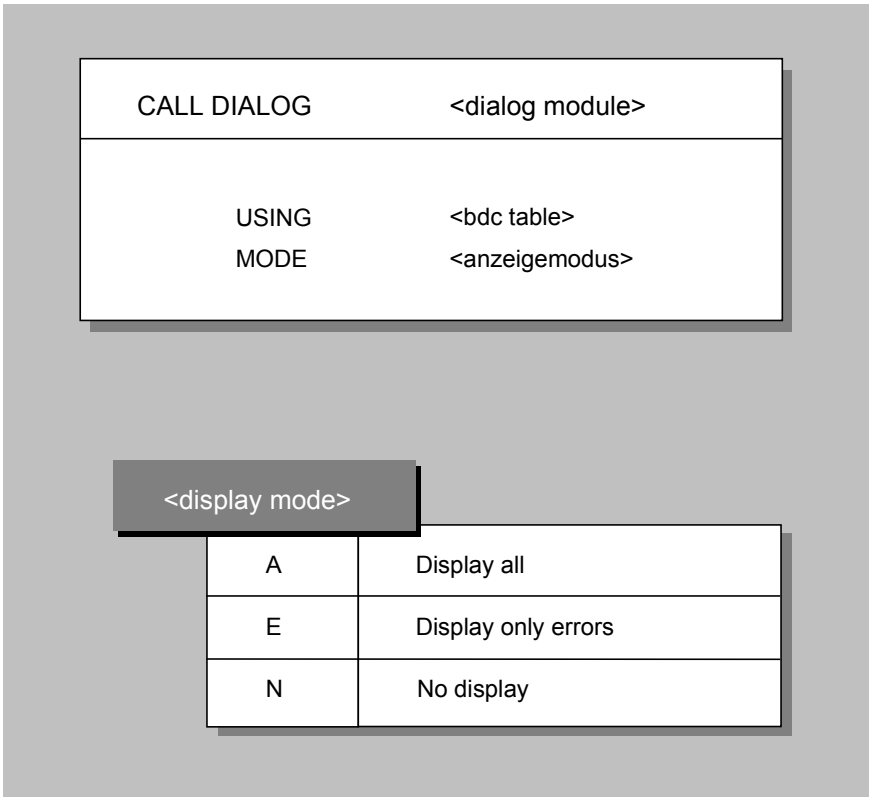
  Alternatively, you can schedule both the batch input program and RSBDCSUB as job steps in a single background job. In this case, however, RSBDCSUB is started even if the batch input program should terminate abnormally.

For detailed information about processing batch input sessions, see Managing Batch Input Sessions [Ext.] in the *System Services* guide.

Using CALL DIALOG with Batch Input

A program that uses CALL DIALOG to submit batch input should perform the following steps:

1. For the screens you want processed, fill all desired fields.
2. Use a CALL DIALOG statement to call the dialog module and to pass it to the BDCDATA table.



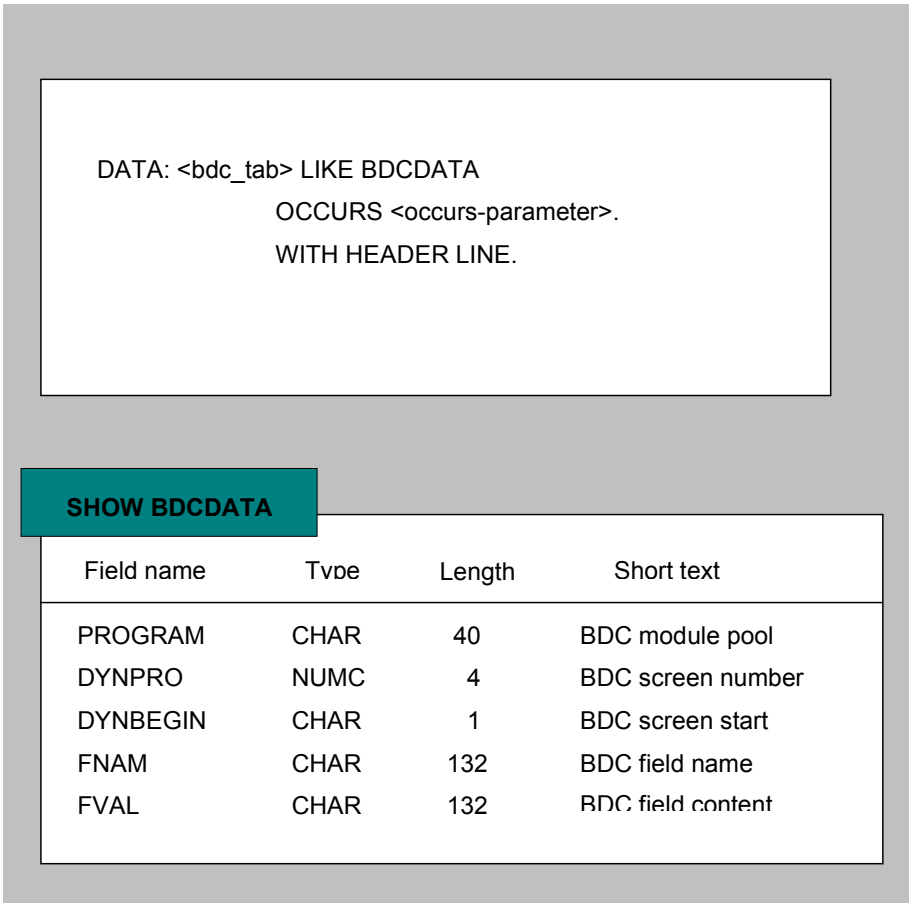
The MODE parameter in the CALL statement is explained in Using CALL TRANSACTION USING for Batch Input [Page 132]

```
CALL DIALOG 'DIALOG1' USING BDCDATA MODE 'E'.
```

Using the Data Transfer Data Structure

If you want to write data to a batch input session or to process the data using **CALL TRANSACTION USING** or **CALL DIALOG**, you must prepare an internal table **<bdc_tab>**. This internal table must be structured according to ABAP Dictionary structure **BDCDATA.** The internal table **<bdc_tab>** stores the data that is to be entered into R/3 System and the actions that are necessary to process the data. You can think of the table as storing the script that the R/3 System is to follow in processing batch input data.

The graphic below shows how to declare the internal table **<bdc_tab>** in your ABAP program and the fields contained in the structure BDCDATA.



Process flow

1. Declare internal table **<bdc_tab>**.
2. Initialize the internal table before you call each new transaction.

Process flow

3. At the beginning of each new screen, you must maintain the module pool name **<program>**, the screen number **<dynpro>** and a flag:

```
<bdc_tab>-PROGRAM = <program>.
<bdc_tab>-DYNPRO = <dynpro>.
<bdc_tab>-DYNBEGIN = 'X'.
APPEND <bdc_tab>.
```

4. For each field to which you want to assign values, insert an entry in the internal table. Specify the technical field name **<fnam>** and the field content **<fval>**:

```
<bdc_tab>-FNAM = <fnam>.
<bdc_tab>-FVAL = <fval>.
APPEND <bdc_tab>.
```

   If the field is in a step loop or a table control, you must also specify the lines in which the input is to be entered. The field name extension displays the line number:

```
<bdc_tab>-FNAM = 'fieldx(5)'.
```

5. If you want to position the cursor on a particular field, enter the cursor position by filling field **FNAM** with the value **BDC_CURSOR,** and transferring into the field **FVAL** the technical name **<tname>** of the field where the cursor should be:

```
<bdc_tab>-FNAM = 'BDC_CURSOR'.
<bdc_tab>-FVAL = <tname>.
APPEND <bdc_tab>.
```

   If you want to position the cursor on a field in a step loop or table control, you must also specify the lines in which the input is to be entered. The field name extension displays the line number:

```
<bdc_tab>-FVAL = 'fieldx(5)'.
```

6. Now specify which action is to be executed in this screen. You must determine the triggered function code **<fcode>** and assign this with the field **FVAL**. Note that the character '**/**' should always be placed before the function key number. The character '**=**' must be placed before all other function codes.
   Assign value **BDC_OKCODE** to the field **FNAM**:

```
<bdc_tab>-FNAM = 'BDC_OKCODE'.
<bdc_tab>-FVAL = <fcode>.
APPEND <bdc_tab>.
```

7. Execute steps 3 to 6 for each additional screen in the transaction.

8. After the last screen in the transaction, internal table **<bdc_tab>** is filled with all of the values required. You can now use this table to create an entry in a batch input session or to call the commands **CALL TRANSACTION** or **CALL DIALOG**.
   The transaction to which a BDCDATA structure refers is identified separately. If your program writes data to a batch input session, then the transaction is specified in the call to the BDC_INSERT function module. This function module writes a BDCDATA structure out to the session. If your program processes data with CALL TRANSACTION USING, then the transaction is specified directly in this statement.

The following table shows what the contents of a BDCDATA structure might look like. This BDCDATA structure would add a line to a report in Transaction *SE38*, the ABAP Editor:

BDCDATA Structure for Adding a Line to a Report (Transaction SE38)

PROGRAM	DYNPRO	DYNBEGIN	FNAM	FVAL
SAPMS38M	0100	X		
			RS38M-PROGRAMM	<Name>
			RS38M-FUNC_EDIT	X
			BDC_OKCODE	=CHAP (Change function code)
SAPMSEDT	2310	X		
			RSTXP-TDLINECOM(1)	B-
SAPMSEDT	2310	X		
			BDC_CURSOR	RSTXP-TDLINECOM(1)
			RSTXP-TDLINE(1)	BDC Test Text
			BDC_OKCODE	/11 (Save function key)
SAPMSEDT	2310	X		
			BDC_OKCODE	/3 (Back function key)
SAPMS38M	0100	X		
			BDC_OKCODE	/15 (Quit function key)

Determining System Data

You require system data to manually program or modify batch input programs. For screen sequence control of the transaction, you need the program name and screen number. You also need the function code to display the next screen.

To fill a field you require its technical field name.

To determine the system data, open an additional R/3 session. Execute the transaction in this session, until the system displays the screen that you require for the system data.

Determining Program Names and Screen Numbers:

On the maintenance screen of the transaction, choose *System* → *Status...*. The system displays the relevant system information. The program name is specified in the field *Program (screen)* and the screen number is displayed in the field *Screen number*.

If the maintenance screen is a modal dialog box, you can display the system information by positioning the cursor on a field in the dialog box and choosing `F1`. The system displays the help text in an additional dialog box. Choose *Technical info.* The information you require is displayed in the fields *Program name* and *Screen number*.

Determining Function Codes:

If a function is assigned a function key, you can display this by clicking with the right mouse button in the work area.

If a function has not been assigned a function key, use the left mouse button to choose that function. With this button still pressed, choose the function key `F1`. The system displays a dialog box that displays the function code in the field *Function*.

Determining Field Names:

- Position the cursor on the field and choose F1 help.
- The system displays a dialog box. Choose *Technical info*.
- The subsequent dialog box displays the information you require in the *Screen field*.

Frequent Data Transfer Errors

The most frequent errors include:

- The BDCDATA structure contains screens in incorrect sequence.
- The BDCDATA structure assigns a value to a field that does not exist on the current screen.
- The BDCDATA structure contains a field that exceeds the specified length.

General guidelines

You should be aware of the following guidelines when you create sessions and call transactions or dialogs:

- You must provide data for all required fields on a screen.
- You can only specify the initial data for a screen. The system does not accept input as a response to a warning or an error message.
- If there is more than one possible screen sequence for a transaction or dialog, your program specifies the screen sequence for the transaction. You must transfer all screens that the dialog user sees to the selected screen sequence. This applies even if the screen itself is not used to input data.

Direct Input

To enhance the batch input procedure, the system offers the direct input technique, especially for transferring large amounts of data. In contrast to batch input, this technique does not create sessions, but stores the data directly. It does not process screens. To enter the data into the corresponding database tables directly, the system calls a number of function modules that execute any necessary checks. In case of errors, the direct input technique provides a restart mechanism. However, to be able to activate the restart mechanism, direct input programs must be executed in the background only. To maintain and start these programs, use program RBMVSHOW or Transaction *BMV0*.

Examples for direct input programs are:

Program	Application
RFBIBL00	FI
RMDATIND	MM
RVAFSS00	SD
RAALTD11	AM
RKEVEXT0	CO-PA

For more information, see the respective program documentation.

Programming with External Commands

This section describes the interface for calling external commands from ABAP programs. The interface offers authorization protection against misuse of OS commands.

For information on maintaining external command definitions, please see the *Computing Center Management System* guide.

Programming Techniques

There are three programming techniques that are likely to be useful in application programs:

- Execute an external command: express method.

  With SXPG_CALL_SYSTEM, you can check the user's authorization for the specified command and run the command. The command runs on the host system on which the function module is executed. The function module is RFC-capable. It can therefore be run on the host system at which a user happens to be active or on another designated host system at which an R/3 server is active.

- Read external operation system commands into internal tables:

  Function modules:

  – SXPG_COMMAND_LIST_GET: Select a list of external command definitions.

  – SXPG_COMMAND_DEFINITION_GET: Read the definition of a single external command from the R/3 System's database.

- Execute an external command after first checking the user's authorization to use the command.

  Function modules:

  – SXPG_COMMAND_CHECK: Check whether the user is authorized to execute the specified command on the target host system with the specified arguments.

  – SXPG_COMMAND_EXECUTE: Check a user's authorization to use a command, as in SXPG_COMMAND_CHECK. If the authorization check is successful, then execute the command on the target host system.

Maintaining external commands should be carried out by way of the standard R/3 function (transaction SM69), not with additional programs that use the maintenance function modules. You should also note that the Computing Center Management System) offers a standard transaction for running an external command (transaction SM49).

SXPG_CALL_SYSTEM: Run an External Command (Express Method)

Use this function module to do the following:

- check a user's authorization to run a command
- carry out the command.

To determine the system on which the command should be carried out, the function module takes by default the user's current host system and the operating system type.

Since the function module is RFC-capable, you can use RFC (the remote function call interface) to carry run the function module at another R/3 application server. The external command is then carried out on the host system on which the destination server is running. See CALL FUNCTION... DESTINATION in the ABAP syntax documentation or *User's Guide* for more information.

SXPG_CALL_SYSTEM checks the command that is specified in the call before carrying it out. The function module follows these rules in performing this check:

- It checks for a command definition that has the same operating system type as in the system field SY-OPSYS. If found, then this command definition is used to run the command.
- If the first check fails, then the function module looks to see if the command has been defined with an operating system type that belongs to the same syntax group as the SY-OPSYS operating system. A syntax group is an R/3 construct that groups together OS's that share the same command and file name syntax. Example: the various flavors of UNIX belong to a syntax group.

  If found, then this command definition is used to run the command. Example: If SY-OPSYS shows HP-UX as the OS, then a command definition for Sun UNIX would also be acceptable for carrying out the command.
- If the second check also fails, then the function modules looks for the command with the operating system type ANYOS. ANYOS indicates that a command definition can be carried out on all supported host operating systems.

  If found, then this command definition is used to run the command.

Syntax:

```
CALL FUNCTION 'SXPG_CALL_SYSTEM'
     IMPORTING
        COMMANDNAME    = <Name of command to run>  " Default '*'
        PARAMETERS     = <Argument string>  " Default <space>
     EXPORTING
        STATUS         = <Exit status of command>
      TABLES
        EXEC_PROTOCOL = <Log>  " In structure BTCXPM.  Can
                               " contain STDOUT, STDERR
      EXCEPTIONS
        NO_PERMISSION " Command rejected by user exit auth.
                      " check
        COMMAND_NOT_FOUND  " Command not defined in R/3 database
        PARAMETERS_TOO_LONG  " Complete parameter string exceeds
                             " 128 characters
```

```
SECURITY_RISK " Security check failed
WRONG_CHECK_CALL_INTERFACE " Problem with function
                           " module for additional
                           " security check
PROGRAM_START_ERROR " Error while starting program
PROGRAM_TERMINATION_ERROR " Error while requesting final
                          " status of program
X_ERROR " Reserved
PARAMETER_EXPECTED " Required parameter not specified
TOO_MANY_PARAMETERS " User arguments not allowed by
                    " supplied in call
ILLEGAL_COMMAND " Command not legitimately defined
OTHERS
```

Parameters

IMPORTING Parameters

Parameter name	Use
COMMANDNAME	The name of the definition of the external command, as specified in the maintenance function (transaction SM69).
PARAMETERS	Arguments for the external command as specified by the definition in the R/3 System and by the calling program or user. These arguments are checked for impermissible characters, such as the ; under UNIX. Problems are registered with the SECURITY_RISK exception.

EXPORTING Parameters

Parameter name	Use
STATUS	Returns the final status of the execution of the external command: • Value ‘O’: The external command was started and ran to end successfully. • Value ‘E’: An error occurred; the external command was not run successfully.

Tables Parameters

Parameter name	Use
EXEC_PROTOCOL	Contains the STDOUT and STDERR output of the external command and any output from the target host system.

Exceptions

Exception name	Meaning
X_ERROR	Reserved for future use.
NO_PERMISSION	The AUTHORITY-CHECK of the user's authorization for the authorization object S_LOG_COM failed. The user is not authorized to carry out the command named with the specified arguments on the target system.
COMMAND_NOT_FOUND	Command name, as identified by COMMANDNAME and OPERATINGSYSTEM, has not been defined in the maintenance function (transaction SM69).
PARAMETERS_TOO_LONG	The combined argument string (ADDITIONAL_PARAMETERS and the DEFINED_PARAMETERS, as returned in ALL_PARAMETERS) exceeds the limit of 128 characters in length.
SECURITY_RISK	Either: • The command contains impermissible characters. These are characters with potentially dangerous properties, such as ; under UNIX. • The command definition specifies that an extra-check function module be run. This function module has rejected execution of the command.
WRONG_CHECK_CALL_ INTERFACE	The command definition specifies that an extra-check function module is to be run. Either this function module is missing, or the interface defined for this function module does not match that of the standard R/3 function module SXPG_DUMMY_COMMAND_CHECK. For more information, please see SXPG_DUMMY_COMMAND_CHECK: Interface for Extra-Check Function Modules [Page 167].
TOO_MANY_PARAMETERS	The command definition specifies that user-specified arguments for the external command are not allowed. However, an additional string of command arguments was specified.
PARAMETER_EXPECTED	The command definition includes the placeholder character ?, which signifies that additional user-defined arguments are required. However, no additional arguments string was supplied.
PROGRAM_START_ERROR	An error occurred while starting the external command. The R/3 system field SY-MSGV1 contains additional information on the problem.
PROGRAM_TERMINATION_ ERROR	An error occurred while trying to obtain the return code of the external program. The R/3 system field SY-MSGV1 contains additional information on the problem.

ILLEGAL_COMMAND	The external command definition was modified “illegally”. That is, the command was not modified by means of the maintenance function (transaction SM69). The modified command is registered in the system log in its substituted form. The message is registered under the system log ID “LC”.
OTHERS	Catch any new exceptions added to this function module.

SXPG_COMMAND_EXECUTE: Check Authorization for and Execute an External Command

Use this function module to check a user's authorization to run a particular external command and then carry out the command.

Like SXPG_COMMAND_CHECK, this function module checks that the user is authorized to execute the command:

- with the arguments specified in ADDITIONAL_PARAMETERS
- on the target host system, as identified by OPERATINGSYSTEM and TARGETSYSTEM.

If any R/3 profile parameter has been inserted in the portion of the command stored in the database, then the value of this parameter is substituted into the command. If an R/3 application server is active on the target system (TARGETSYSTEM), then the profile parameter values are read from the profile in effect on that system. No parameter substitution is made in ADDITIONAL_PARAMETERS.

After substitution, the command is checked for the presence of "dangerous" characters, such as the semicolon ; on UNIX systems.

If an additional "safety" function module has been specified in the definition of the command, then this is called as well in the course of processing. This function module can prevent execution of the command.

If the authorization checks complete successfully, then the command is run on the target host system.

Syntax:

```
CALL FUNCTION 'SXPG_COMMAND_EXECUTE'
     IMPORTING
        COMMANDNAME     = <command name in R/3 System>
        OPERATINGSYSTEM = <external command name>
                              " Default SY-OPSYS
        TARGETSYSTEM    = <Target host name> " Default SY-HOST
        STDOUT          = 'X' " Log STDOUT if marked. Default
                              " 'X'
        STDERR          = 'X' " Log STDERR if marked.  Default
                              " 'X'
        TERMINATIONWAIT = 'X' " Synchronous program start,
                              " Wait for termination and
                              " event log, if returned.
                              " DEFAULT 'X'
        TRACE           = ' ' " Trace execution.  Unmarked, no
                              " trace. If value TRACE_LEVEL3,
                              " trace active
        ADDITIONAL_PARAMETERS = <user-specified argument string>
                              " DEFAULT SPACE
        ABAPPROG = <ABAP program> " Default space
        ABAPFORM = <ABAP form in program> " Default space
        JOBCOUNT = <Job count> " Reserved for future use.
                               " Default space
      EXPORTING
```

SXPG_COMMAND_EXECUTE: Check Authorization for and Execute an External Command

```
      STATUS = <Exit status of command>
   TABLES
      EXEC_PROTOCOL = <Log>  " In structure BTCXPM.  Can
                             " contain STDOUT, STDERR
   EXCEPTIONS
      NO_PERMISSION " Command rejected by user exit auth.
                    " check
      COMMAND_NOT_FOUND " Command not defined in R/3 database
      PARAMETERS_TOO_LONG " Complete parameter string exceeds
                          " 128 characters
      SECURITY_RISK " Security check failed
      WRONG_CHECK_CALL_INTERFACE " Problem with function
                                 " module for additional
                                 " security check
      PROGRAM_START_ERROR " Error while starting program
      PROGRAM_TERMINATION_ERROR "Error while requesting final
                                " status of program
      X_ERROR " Reserved
      PARAMETER_EXPECTED " Required parameter not specified
      TOO_MANY_PARAMETERS " User arguments not allowed by
                          " supplied in call
      ILLEGAL_COMMAND " Command not legitimately defined
      WRONG_ASYNCHRONOUS_PARAMETERS " Reserved for future
                                    " use
      CAT_ENQ_TBTCO_ENTRY " Reserved for future use
      JOBCOUNT_GENERATION_ERROR " Reserved for future use
      OTHERS.
```

Parameters

IMPORTING Parameters

Parameter name	Use
COMMANDNAME	The name of the definition of the external command, as specified in the maintenance function (transaction SM69).
OPERATING SYSTEM and TARGETSYSTEM	Identify the host system on which the command is to be executed. OPERATINGSYSTEM is specified in the command definition (transaction SM69). TARGETSYSTEM is the host name of the system upon which the command is to run
ADDITIONAL_PARAMETERS	Arguments for the external command as specified by the calling program or user. These arguments are appended to any arguments specified in the externalcommand definition (DEFINED_PARAMETERS). These arguments are checked for impermissible characters, such as the ; under UNIX. Problems are registered with the SECURITY_RISK exception.

STDOUT	Log STDOUT output from the external command in EXEC_PROTOCOL, if set to a non-space value. If set to space, then STDOUT is ignored. Can be logged only if TERMINATIONWAIT has a non-space value (wait for termination).
STDERR	Log STDERR output from the external command in EXEC_PROTOCOL, if set to a non-space value. If set to space, then STDERR is ignored. Can be logged only if TERMINATIONWAIT has a non-space value (wait for termination).
TERMINATIONWAIT	Wait for termination of external command. If set to space ' ', then the command is started asynchronously and no output is collected from the command or from the target host system. If set to a value other than space, then the function module waits for the external command to complete. It also logs STDOUT and STDERR, if requested, in EXEC_PROTOCOL, if the TARGETSYSTEM returns this output.
TRACE	Trace execution through CALL 'writetrace' and through the local trace function of the external commands interface itself. If set to space ' ', then no trace is carried out. Otherwise, tracing is active. Should be used ONLY for testing. The setting of this argument has no effect upon the trace specification in the definition of the external command (transaction SM69).

EXPORTING Parameters

Parameter name	Use
STATUS	Returns the final status of the execution of the external command: • Value 'O': The external command was started and ran to end successfully. • Value 'E': An error occurred; the external command was not run successfully.

Tables Parameters

Parameter name	Use

EXEC_PROTOCOL	Contains the STDOUT and STDERR output of the external command and any output from the target host system, if TERMINATIONWAIT is activated.

Exceptions

Exception name	Meaning
X_ERROR	Reserved for future use.
NO_PERMISSION	The AUTHORITY-CHECK of the user's authorization for the authorization object S_LOG_COM failed. The user is not authorized to carry out the command named with the specified arguments on the target system.
COMMAND_NOT_FOUND	Command name, as identified by COMMANDNAME and OPERATINGSYSTEM, has not been defined in the maintenance function (transaction SM69).
PARAMETERS_TOO_LONG	The combined argument string (ADDITIONAL_PARAMETERS and the DEFINED_PARAMETERS, as returned in ALL_PARAMETERS) exceeds the limit of 128 characters in length.
SECURITY_RISK	Either: • The command contains impermissible characters. These are characters with potentially dangerous properties, such as ; under UNIX. • The command definition specifies that an extra-check function module be run. This function module has rejected execution of the command.
WRONG_CHECK_CALL _INTERFACE	The command definition specifies that an extra-check function module is to be run. Either this function module is missing, or the interface defined for this function module does not match that of the standard R/3 function module SXPG_DUMMY_COMMAND_CHECK. For more information, please see SXPG_DUMMY_COMMAND_CHECK: Interface for Extra-Check Function Modules [Page 167].
TOO_MANY_PARAMETERS	The command definition specifies that ADDITIONAL_PARAMETERS are not allowed. However, an additional string of command arguments was specified.
PARAMETER_EXPECTED	The command definition includes the placeholder character ?, which signifies that ADDITIONAL_PARAMETERS is required. However, no additional arguments string was supplied.

PROGRAM_START_ERROR	An error occurred while starting the external command. The R/3 system field SY-MSGV1 contains additional information on the problem.
PROGRAM_TERMINATION_ ERROR	An error occurred while trying to obtain the return code of the external program. The R/3 system field SY-MSGV1 contains additional information on the problem.
ILLEGAL_COMMAND	The external command definition was modified “illegally”. That is, the command was not modified by means of the maintenance function (transaction SM69). The modified command is registered in the system log in its substituted form. The message is registered under the system log ID “LC”.
WRONG_ASYNCHRONOUS_ PARAMETERS	Reserved for future use.
CAT_ENQ_TBTCO_ENTRY and JOBCOUNT_GENERATION _ERROR	Reserved for future use.
OTHERS	Catch any new exceptions added to this function module.

SXPG_COMMAND_CHECK: Check Authorization to Execute an External Command

Use this function module to check a user's authorization to execute a particular external command. The function module checks that the user is authorized to execute the command:

- with the arguments specified in ADDITIONAL_PARAMETERS
- on the target host system, as identified by OPERATINGSYSTEM and TARGETSYSTEM.

If any R/3 profile parameter has been inserted in the portion of the command stored in the database, then the value of this parameter is substituted into the command. If an R/3 application server is active on the target system (TARGETSYSTEM), then the profile parameter values are read from the profile in effect on that system. No parameter substitution is made in ADDITIONAL_PARAMETERS.

After substitution, the command is checked for the presence of "dangerous" characters, such as the semicolon ; on UNIX systems.

If an additional "safety" function module has been specified in the definition of the command, then this is called as well in the course of processing. This function module can prevent execution of the command.

Should the function module complete successfully (without exceptions), then the following is returned:

- The complete command name (including the path) from the command definition. R/3 profile parameters in the command are resolved to their values.
- The command arguments from the command definition. R/3 profile parameters in the string are resolved to their values.
- the additional parameters string, as specified in the call to the function module.

Profile parameters are resolved once again to their values when a command is actually dispatched to run.

Syntax

```
CALL FUNCTION 'SXPG_COMMAND_CHECK'
  IMPORTING
    ADDITIONAL_PARAMETERS = <Argument string> " Default <space>
    COMMANDNAME = <Name of command definition in R/3 System>
    OPERATINGSYSTEM = <Target OS as defined in R/3 System>
    TARGETSYSTEM = <Host system for execution of command>
  EXPORTING
    ALL_PARAMETERS = <Complete argument string>
    DEFINED_PARAMETERS = <Arguments from command definition>
    PROGRAMNAME = <Complete pathname of OS command>
  EXCEPTIONS
    X_ERROR  " Reserved
    COMMAND_NOT_FOUND  " Command not defined in R/3 database
    NO_PERMISSION " Command rejected by user exit auth. check
    PARAMETERS_TOO_LONG  " Complete parameter string exceeds
                         " 128 characters
    PARAMETER_EXPECTED  " Required parameter not specified
```

```
SECURITY_RISK  " Security check failed
TOO_MANY_PARAMETERS  " No ADDITONAL_PARAMETERS allowed
WRONG_CHECK_CALL_INTERFACE  " Problem with function module
                            " for additional security check
ILLEGAL_COMMAND  " Command not legitimately defined
OTHERS.
```

Parameters

IMPORTING Parameters

Parameter name	Use
ADDITIONAL_PARAMETERS	Arguments for the external command as specified by the calling program or user. These arguments are appended to any arguments specified in the externalcommand definition (DEFINED_PARAMETERS). These arguments are checked for impermissible characters, such as the ; under UNIX. Problems are registered with the SECURITY_RISK exception.
COMMANDNAME	The name of the definition of the external command, as specified in the maintenance function (transaction SM69).
OPERATINGSYSTEM and TARGETSYSTEM	Identify the host system on which the command is to be executed. OPERATINGSYSTEM is specified in the command definition (transaction SM69). TARGETSYSTEM is the host name of the system upon which the command is to run.

EXPORTING Parameters

Parameter name	Use
ALL_PARAMETERS	Returns the complete argument string for the command, consisting of ADDITIONAL_PARAMETERS and DEFINED_PARAMETERS. The string is as close to the potential runtime string as possible. For example, variables are substituted into the string from the target host system, if possible.
DEFINED_PARAMETERS	Returns the predefined argument string from the command definition in the R/3 System. The string is as close to the potential runtime string as possible. For example, variables are substituted into the string from the target host system, if possible.
PROGRAMNAME	The complete name (including path) of the command that is to be executed. Arguments are not included.

Exceptions

Exception name	Meaning
X_ERROR	Reserved for future use.
NO_PERMISSION	The AUTHORITY-CHECK of the user's authorization for the authorization object S_LOG_COM failed. The user is not authorized to carry out the command named with the specified arguments on the target system.
COMMAND_NOT_FOUND	Command name, as identified by COMMANDNAME and OPERATINGSYSTEM, has not been defined in the maintenance function (transaction SM69).
PARAMETERS_TOO_LONG	The combined argument string (ADDITIONAL_PARAMETERS and the DEFINED_PARAMETERS, as returned in ALL_PARAMETERS) exceeds the limit of 128 characters in length.
SECURITY_RISK	Either: • The command contains impermissible characters. These are characters with potentially dangerous properties, such as ; under UNIX. • The command definition specifies that an extra check function module be run. This function module has rejected execution of the command.
WRONG_CHECK_CALL_INTERFACE	The command definition specifies that an extra check function module is to be run. Either this function module is missing, or the interface defined for this function module does not match that of the standard R/3 function module SXPG_DUMMY_COMMAND_CHECK. For more information, please see SXPG_DUMMY_COMMAND_CHECK: Interface for Extra-Check Function Modules [Page 167].
TOO_MANY_PARAMETERS	The command definition specifies that ADDITIONAL_PARAMETERS are not allowed. However, an additional string of command arguments was specified.
PARAMETER_EXPECTED	The command definition includes the placeholder character ?, which signifies that ADDITIONAL_PARAMETERS is required. However, no additional arguments string was supplied.
ILLEGAL_COMMAND	The external command definition was modified "illegally". That is, the command was not modified by means of the maintenance function (transaction SM69).
OTHERS	Catch any new exceptions added to this function module.

SXPG_COMMAND_LIST_GET: Read a List of External Commands

Use this function module to read a list of the external commands that have been defined in your R/3 System into an internal table. You can then loop over the table to select a command, or offer the list to your user for selection. You can then pass the selection on to SXPG_COMMAND_EXECUTE for an authorization check and execution of the command.

Syntax:

```
CALL FUNCTION 'SXPG_COMMAND_LIST_GET'
     IMPORTING
        COMMANDNAME       = <R/3 command name>  " Default '*'
        OPERATINGSYSTEM   = <OS type in command definition>
                                                " Default '*'
        TARGETSYSTEM      = <Host name for running command>
                                                " Default '*'
     TABLES
        COMMAND_LIST      = <Command list>      " Structure
                                                " SXPGCOLIST
     EXCEPTIONS
        OTHERS = 1.
```

Parameters

IMPORTING Parameters

Parameter name	Use
COMMANDNAME	The name of the definition of the external command, as specified in the maintenance function (transaction SM69). Generic searching: You can use the * wild-card character to represent any number of occurrences of any character. You can use * anywhere in the value.
OPERATINGSYSTEM	The type of operating system upon which a command is to be carried out. Use to restrict the commands that are selected to the operating system of interest. “ANYOS” selects R/3 standard commands that can run be run as they are defined on any operating system. Generic searching: You can use the * wild-card character to represent any number of occurrences of any character. You can use * anywhere in the value.
TARGETSYSTEM	Host name of the system upon which the selected command is to be carried out. Optional: Use this parameter to have the system check the user’s authorization to run commands in the target system. If the user is authorized, then COMMAND_LIST-PERMISSION is set to ‘X’. Otherwise, the field is set to space ‘ ’.

TABLES Parameters

Parameter name	Use
COMMAND_LIST	Contains a list of the selected commands in the format shown in transaction SM49 or SM69. The user can select a command from the list to run, if you pass the user's selection on to SXPG_COMMAND_EXECUTE. The table is empty if no command is found that meets the selection criteria. No exception is raised in this case. Notes on fields: • COMMAND_LIST-PERMISSION shows whether the calling user is authorized to run a command on the system that you name in TARGETSYSTEM (optional). `'X'` means that the user is authorized, `' '` means that the user does not have authorization to run commands on TARGETSYSTEM. • COMMAND_LIST-TRACEON and COMMAND_LIST-CHECKALL are set to `'X'` if the user has the authorization to maintain external commands. • COMMAND_LIST-SAPCOMMAND is set to `'X'` if the command in that row is an R/3 standard command. These commands may not be modified in customer systems.

SXPG_COMMAND_DEFINITION_GET: Read Single External Command

Use this function module to read the definition of a particular external command into an internal table.

You must set the COMMANDNAME and OPERATINGSYSTEM parameters to the appropriate values before you call the function module.

Syntax:

```
CALL FUNCTION 'SXPG_COMMAND_DEFINITION_GET'
     IMPORTING
        COMMANDNAME     = <Name of command in R/3 System>
        OPERATINGSYSTEM = <Target OS as defined in R/3 System>
        TARGETSYSTEM    = <Host system for running command>
                          " Default '*'
      EXPORTING
        COMMAND         = <Command definition> " Structure
                                               " SXPGCOLIST
      EXCEPTIONS
        COMMAND_NOT_FOUND " Command not defined in R/3 database
        OTHERS.
```

Parameters

IMPORTING Parameters

Parameter name	Use
COMMANDNAME	The name of the definition of the external command, as specified in the maintenance function (transaction SM69). Must be set before you call the function module.
OPERATINGSYSTEM	The type of operating system upon which a command is to be carried out. Must be set before you call the function module.
TARGETSYSTEM	Host name of the system upon which the selected command is to be carried out. Optional: use to have the system check on the user's authorization to run commands on the target system. If the user is authorized, then COMMAND_LIST-PERMISSION is set to `'X'`. Otherwise, the field is set to space `' '`.

EXPORTING Parameters

Parameter name	Use

COMMAND	Contains the definition of the selected command in the format shown in transaction SM49 or SM69 (structure SXPGCOLIST). The user can have the command run, if you pass the user's selection on to SXPG_COMMAND_EXECUTE. If no command is found, then the exception COMMAND_NOT_FOUND is triggered. Notes on fields: • COMMAND_LIST-PERMISSION shows whether the calling user is authorized to run a command on the system that you name in TARGETSYSTEM (optional). `'X'` means that the user is authorized, `' '` means that the user does not have authorization to run commands on TARGETSYSTEM. • COMMAND_LIST-TRACEON and COMMAND_LIST-CHECKALL are set to `'X'` if the user has the authorization to maintain external commands. • COMMAND_LIST-SAPCOMMAND is set to `'X'` if the command in that row is an R/3 standard command. These commands may not be modified in customer systems.

SXPG_DUMMY_COMMAND_CHECK: Interface for Extra-Check Function Modules

When you define an external command, you can specify that an additional function module be called to check a user's authorization to run a command.

You can define such an additional authorization check module yourself. Your function module must have the interface defined below for SXPG_DUMMY_COMMAND_CHECK. Like this function module, it may only allow or deny permission to carry out the command.

To ensure that such a user exit function module cannot be misused, the system checks that the interface of the authorization-checking function module matches the interface of SXPG_DUMMY_COMMAND_CHECK. It is therefore essential that SXPG_DUMMY_COMMAND_CHECK never be changed.

To write your own authorization-checking module, do the following:

1. Copy SXPG_DUMMY_COMMAND_CHECK to a name in the customer naming range (names beginning with Y or Z).
2. Write your authorization check.
3. Specify that the function module should be run when the command that is to be checked is run. Do this in the command definition (transaction SM69).

Do not change SXPG_DUMMY_COMMAND_CHECK or its interface. Never call this function module, either directly or by specifying it as the check module in a command definition. Copy it and then modify the new function module in order to implement an additional authorization check.

Interface Syntax:

```
IMPORTING
     PROGRAMNAME = <Program or command from definition>
     PARAMETERS  = <Argument string>
EXCEPTIONS
     NO_PERMISSION " Command rejected by user exit auth. check
```

Parameters

IMPORTING Parameters

Parameter name	Use
PROGRAMNAME	The fully substituted pathname of the external command that is to be run.
PARAMETERS	Arguments for the external command as specified by the definition in the R/3 System and by the calling program or user. These arguments are checked for impermissible characters, such as the ; under UNIX. Problems are registered with the SECURITY_RISK exception.

Exceptions

Exception name	Meaning
NO_PERMISSION	Prevents a command from being run if the authorization check that you have implemented in a check module fails. NO_PERMISSION triggers the exception SECURITY_RISK in the calling function module.

Common Application Interfaces

You can find information on other programming interfaces in the document *Extended Function Library*.

The interfaces described in this document are as follows:

- function modules for programming dialog windows
- function modules for building view functionality into your applications
- address maintenance interface
- factory calendar interface
- function modules for converting currencies, weights, and measures
- function modules for creating change documents
- function modules for creating platform-independent file names
- function modules for using number ranges
- archiving system interface