INTRODUCTION

This document was created to outline several naming standards, programming standards, and best practices in ABAP programming and provide guidelines for ABAP development for MIT. These are not "The Official ABAP/4 Standards", however they do contain MIT-specific programming conventions and techniques.

The topics included are general coding standards, performance standards, naming conventions, tips & tricks, and sample programs. As standards are developed, this guide will be updated on an ongoing basis.

The document is intended for use by ABAP developers as a guide for MIT-specific developments and enhancements to R/3. The user of this document should be familiar with basic concepts of R/3 and should have taken a training class equivalent to SAP20 R/3 Overview.

To request clarification of these issues or to suggest other topics to be covered, please contact David Rosenberg (rosenberg@mit.edu).

I. HOW TO UPDATE THIS GUIDE

If you have questions about topics in this guide, would like to suggest changes, or provide new topics to be included, please submit the topics and details about the updates to David Rosenberg (rosenberg@mit.edu).

Revision History

Revision	Date	Description of Changes
1	05/15/98	Initial (post-review) distributed version
2	05/22/98	Document layout changes and spelling corrections
3	9/17/98	Performance Section update, New Tips & Tricks, Application abbreviations added (PM, RQ, BP), New document layout and navigation features. Note that new or modified sections are highlighted in the table of contents with a .

II. ABAP/4 BACKGROUND

The ABAP/4 language is an "event driven", "top-down" programming language. The execution of an event is controlled by the ABAP/4 processor. For example, the event AT SELECTION-SCREEN is executed when the user hits ENTER on the selection screen and the event START-OF-SELECTION is executed when the user presses ENTER to start the program. Because the ABAP/4 language incorporates many "event" keywords and these keywords need not be

in any specific order in the code, it is wise to implement in-house ABAP/4 coding standards.

MIT-ABAP Mail List

Anyone doing SAP development at MIT should subscribe to the MIT-ABAP mail list. This includes employees, contractors, consultants, project leaders, and project managers. The list is intended for all development issues including configuration, workflow, report writer, report painter, basis, system administration, documentation, support, training, management, and master data maintenance.

To subscribe to the list, send e-mail to LISTSERV@MITVMA.MIT.EDU with the body of the message containing the single line "SUBSCRIBE MIT-ABAP [Your real name]". (without the quotes or brackets and filling in your name)

III. CODING STANDARDS

General Standards

1. One command per line

Each ABAP/4 command consists of a sentence ending with a period. Multiple commands can be on one line; however, as a standard start each new command on a new line. This will allow for easier deleting, commenting, and debugging.

2. Indented source code

The ABAP/4 editor has a "Pretty Printer" command to indent by 2 positions specific lines of code and add subroutine comments. Event keywords are typically not indented.

```
DATA: BEGIN OF tab OCCURS 100,
        f1 LIKE sg-field1,
        f2 LIKE sg-field2,
      END OF tab.
DATA: f1 TYPE I,
      f2 TYPE I.

START-OF-SELECTION.

GET table.
  IF f1 = f2.
       f1 = 0.
  ELSE.
     f1 = 1.
  ENDIF.

SELECT * FROM tabl WHERE
  field1 EQ sg-field2 AND
  field2 BETWEEN sg-field5 AND sg-field6.
    MOVE ...
    APPEND ...
ENDSELECT.
```

```
END-OF-SELECTION.

LOOP AT tab.
  AT NEW f1.
    CASE F1.
      WHEN ... WRITE:/ ...
      WHEN ... WRITE:/ ...
    ENDCASE.
  ENDAT.
  WRITE:/ f1, f2, ...
ENDLOOP.
```

3. Variable naming

ABAP/4 variable names can be up to 30 characters for DATA fields and subroutines and up to 8 characters for SELECT-OPTIONS and PARAMETERS, therefore, as a standard make the names descriptive. Since SAP segment/table-field names are hyphenated with a '-' (dash), use an '_' (underline) to separate the words for program-specific variables. Whenever possible, the LIKE parameter should be used to define work fields.

```
DATA: MAT_DT LIKE SY-DATUM,
      DAYS TYPE I,
      HOLD_ACCTNO LIKE sg-field1,
      GROSSAMT(10) TYPE P DECIMALS 2,
      GROSS_PERCENT(3) TYPE P DECIMALS 2,
      NET%(3) TYPE P DECIMALS 2.
```

A good rule-of-thumb is to differentiate between DATA fields, SELECT-OPTIONS and PARAMETERS by prefixing selection fields with 'S' or 'P'.

```
SELECT-OPTIONS: S_USER FOR sg-fieldx,
                S_DATE FOR sg-fieldy.

PARAMETERS: P_DEST(4).
```

Subroutine examples:

```
FORM CALCULATE_MATURITY_DATE.
  MAT_DT = SY-DATUM + DAYS.
ENDFORM.
```

or for COBOL-like standards

```
FORM F100_CALCULATE_MATURITY_DATE.
```

4. Reusable code

If a block of code is executed more than once, it should be placed in a subroutine at the bottom of the code. This makes the code more readable, requires less indentation, and is easier to debug since the debugger can jump through an entire subroutine via a PF key. Also, when possible parameters should be passed to and from subroutines to make the purpose easier to understand and reduce the need for global variables. Always document the purpose of each parameter.

5. Parameter passing in subroutines

Whenever possible use a TYPE or LIKE statement when specifying the formal parameters of a subroutine. This is a good programming style, plus it allows

the ABAP compiler to generate more efficient code (which can increase performance up to a factor of 2x). For example:

```
form <subroutine> tables p_tab1   like <tab1>[]
                  using  p_param1 like <data>
                         p_param2 like <dd-field>
                         p_param3 like <dd-structure>
                         p_param4 type <standard data type>
                         p_param5 type <user defined data type>
....
endform.
```

6. Text handling

Literal text which is printed on the report can be handled two ways.

One way is to hard code the literal in the code and give the option of maintaining Text Elements for multi-lingual clients.

```
WRITE:/ 'Grand Total:'(001), ...
```

Another way is to always use Text Elements.

```
WRITE:/ TEXT-001.
```

The advantages to the first way are readability and the Text Element only needs to be maintained for multi-lingual clients. The advantage to the second way is easier maintainability if TEXT-001 is coded several times in the program and it needs to be changed. The editor command REPLACE does not change contents inside single quotes.

7. Program Analysis Utility

To determine the usage of variables and subroutines within a program, you can use the ABAP utility called 'Program Analysis' included in transaction SE38. To do so, execute transaction SE38, enter your program name, then use the path Utilities -> Program Analysis

8. Usage of UserIDs in programs

In no case should a production program or function contain a UserID as either literal or constant data. In the development system it may be necessary to code temporary breakpoints for specific UserIDs, however, these debugging techniques must be removed before the program will be transported.

Authorization Groups

Authorization Groups permit users access to execute report-type programs from transaction SA38 (or SE38). Programs that can be executed by all MIT users must be configured using the Authorization Group 'ZOPN2ALL' for Application '*'. This configuration is done by the developer of the report program in the 'Attributes' screen of SE38.

If a program DOES NOT have the 'ZOPN2ALL' authorization group configured for it, we assume that only certain users should be able to execute it and that other methods of authorization, such as authorization objects are being used to prevent improper use.

Authorizations : Usage of SAP objects

For any custom programming using authorization checks, developers must determine whether a standard SAP authorization object should be used, or if a custom object should be developed. Since the authorization object implementation typically comprises more business-related than technical issues, developers should consult with the business analysts responsible for the application in making this decision.

Dates : Output in reports (Year 2000 compliance)

The preferred formats for representing a date in a printed report are either MM/DD/YYYY or DD-MM-YYYY. In any case, use of a 4-digit year is a requirement.

The R/3 system is already year 2000 compliant. To preserve that compliance, whenever a date is written or read, the year field should contain four digits. The only exception to this standard is reading from or writing to external files, where the year 2000 restriction in the external system may be different. However, even in this case, if it is at all possible, it is desirable to allocate four digits for the year in the file layout.

GUI : SAP Style Guide

For applications that require dialog programming, the SAP Style Guide should be used as a basis for development. Using this guide for screen design and ergonomics will insure consistency in MIT developed transactions. The SAP Style Guide is available in the help documentation using the following path: Help -> R/3 Library -> Basis Components -> ABAP/4 Development Workbench -> SAP Style Guide.

GUI : Screen painter resolution

The MIT user community is, in general, standardized on systems capable of 800x600 resolution. Hence, any screen painter development should insure that all fields are viewable within that screen size. This corresponds to the SAP default screen size of 21 lines by 83 columns. Note that SAP does not include the title, menu bar, status line, and OKCode box in this area.

Interface : Field sizes and layouts in data files

All external file layouts used for interfaces to and from the R/3 system should be discussed with and approved by the Bridges Team before the layout is finalized. This is to insure consistency in field conventions and file processing standards. The requirements for the following data elements are:

Dates

As indicated in the section "Writing dates in reports," dates should use a 4-digit year with certain exceptions.

Cost Objects

Cost objects should be left-aligned in a 12-character field with trailing blanks. (Optionally, a 7-character field followed by 5 characters of blank filler is acceptable)

Cost Elements

Cost elements should be left-aligned in a 10-character field with trailing blanks. (Optionally, a 6-character field followed by 4 characters of blank filler is acceptable)

Interface : Dropbox mechanism for file interfaces

The dropbox is the standard method for data providers to automatically deliver files created outside of SAP as feeds into MITs SAP R/3 environments. Data providers must be registered with the dropbox. These providers FTP data files to the dropbox. Based on a combination of file naming conventions and registered dropbox userids, the files are automatically delivered to the appropriate MIT SAP R/3 environment.

Detailed documentation on how to use this mechanism is located at http://mitvma.mit.edu/~bridges/dropbox.html

Interface : Inbound interface filename determination from event

When an inbound interface is initiated by triggering an SAP event, the filename can be sent as an event parameter which can then be read by the interface program. An example of using this triggering and parameter passing method is shown here:

```
data: infile like rlgrap-filename. " File name pass by event
data: event_flag(1).

* data for getting runtime job info(for event parameter)
data: eventid like tbtcm-eventid,
      eventparm like tbtcm-eventparm,
      external_program_active like tbtcm-xpgactive,
      jobcount like tbtcm-jobcount,
      jobname like tbtcm-jobname,
      stepcount like tbtcm-stepcount.

data: dropbox_dir like rlgrap-filename
        value '/usr/bridges/&func/sapin/'.
…
initialization.
  perform get_event_info.
…
form get_event_info.
  clear event_flag.
  call function 'GET_JOB_RUNTIME_INFO'
```

```
      importing
        eventid = eventid
        eventparm = eventparm
        external_program_active = external_program_active
        jobcount = jobcount
        jobname = jobname
        stepcount = stepcount
      exceptions
        no_runtime_info = 1
        others = 2.
    if sy-subrc = 0.
      event_flag = 'X'.
      if not eventparm is initial.
        clear infile.
        concatenate dropbox_dir eventparm into infile.
      endif.
    endif.
  endform. " GET_EVENT_INFO
```

Jobs : Tracking status using ZJOBRUN

Many programs are inputs or extracts from the R/3 system that require detailed tracking of data file names and error statuses beyond the capability of the standard R/3 job scheduler or if the program is run in the foreground to capture the status its last run. To provide a common repository for job status, the ZJOBRUN table was created. All programs that need to track job status should read / write records to / from this table. Using text files outside of the R/3 system or other custom tables for this purpose is not recommended. Additionally, if there are other programs that are using methods other than the ZJOBRUN table to track status, they should be modified to use ZJOBRUN instead.

ZJOBRUN Table		
FIELD	**DESCRIPTION**	**SAMPLE RECORD**
PGNAM	Program name	ZARI003
BATCH	Sequential number of the job run	85
DATUM	Date of run	05/12/1998
UZEIT	Time of run	19:32:04
RECCOUNT	Record counter – number processed	303

ERRCOUNT	Error Counter – number or error records	42
CNTRLREC	Control Record	261
CREDIT	Total Credits (if applicable)	10,654.04
DEBIT	Total Debits (if applicable)	12,744.79
ERRFILE	Error File – location of error file on UNIX filesystem	/usr/bridges/dev2/saptmp/eards.085.19980512192540
REVIEW	Review Flag	
TEXT	Text – any additional text info	DS cntrlrec/credit = posted
UNAME	User who executed the job	FULLER

Master Data: Searching for a person's name in master data

In SAP master data tables, names can be stored in either upper case or mixed case depending on how they are entered into the system. Any program or function which searches master data tables for a person's name should search in a non-case sensitive fashion to account for this situation.

Updates : Direct database updates of SAP standard tables

Under no circumstances should any program directly update SAP-delivered tables using the INSERT, UPDATE, or DELETE commands. SAP-delivered tables begin with all letters other than Y and Z, and they should only be updated using an SAP transaction. To automate updates to SAP tables via a program, you may only use the CALL TRANSACTION command (or optionally create a BDC using SAP supplied functions).

IV. DOCUMENTATION STANDARDS

Configuration Documentation

FI Document Types (includes AP, GL, MM)

Accounting document type configuration is maintained as part of the SAP configuration process. As such, you must always confirm any new document types with the configuration team prior to coding a report to access the data.

Also, existing extract programs may need to be modified for the new document type, hence it is necessary to pro-actively notify The Financial Architecture Group of the creation of a new document type (Larry Connelly, connelly@mit.edu). After the new document type is approved, the Financial Architecture Group will communicate the changes to all other necessary parties so that other programs can be modified.

Source Code Documentation

ABAP/4 code is fairly self-documenting. However, it is wise to provide future programmers with documentation. Explain the purpose, design, structure and any testing hints at the top of the program. Maintain a history of modification notes, dated, with the most recent change first. Comment work fields, and fields in work records especially those used for interfacing. Comment all subroutines with their purpose. Comments should explain what the code is doing, not how the code is doing it.

MIT has developed several templates for providing documentation via the ABAP 'Program Documentation' function. From the ABAP Editor transaction SE38, you can access the documentation by selecting object component 'Documentation' then click 'Display', or from within the editor screen by doing Goto -> Program doc. The MIT templates are in the following ABAP programs:

ZPROGDOC	**ABAP/4 Program documentation**
ZSYSTDOC	**System documentation**
ZFUNCDOC	**Function Module documentation**
ZINCLDOC	**Include documentation**
ZRPRPDOC	**Report Painter report documentation**

It is mandatory to complete a documentation template prior to transporting your program to the production system.

In addition to documentation via the templates, brief comments within the code can be very helpful for ongoing program

maintenance. For example, you should document the major steps in the program source, such as:

```
do 16 times.  "why are we doing this 16 times instead of 12???
  ...
enddo.
```

For the example of CASE statements, each case value should be documented:

```
case <variable>.
  when 'Y'.  "explain what 'Y' actually means
    ...
  when 'H'.  "explain what 'H' actually means
    ...
endcase.
```

When modifying a program, to document within the source code, you can use the following convention:

```
*---------------------------------------------------------------
---*
* Purpose:
*
*
*---------------------------------------------------------------
---*
* Author:
* Date:
*
*---------------------------------------------------------------
---*
* Modification Log:
*
* 11/2/96 -     Joe Q. Programmer SF2K900002
*               Added .....
* 6/14/96 -     Joe Q. Programmer SF2K900001
*               Changed .....
*---------------------------------------------------------------
---*

DATA: BEGIN OF rec,                                 "SF2K900001
               ind,                                 "interface
type ind.
               f1 LIKE MA-IDNRA,                    "material
number
               f2 LIKE MA-KTX01,                    "material
short text
               f3(16),           "filler            "SF2K900002
                ...
      END OF rec.
```

Always note of which lines were changed or added by appending the transport (change request) number to the line. (This is the convention used by SAP developers)

Function Modules

The import and export parameters of function modules should be documented with brief descriptions of the fields and their typical contents. Also, any special requirements or usage should be noted. At the minimum, the documentation 'Short Text' should be completed for each parameter. This is found by selecting the 'Documentation' radio button on the main function module transaction, SE37.

ABAP/4 Editor

```
INSERT D030L-TYPE D030L-DDNAME D030T-DDTEXT D030L-CBNAME
       D030L-SEGMENTID D030L-DOCTYPE INTO HEADER.
SELECT * FROM D030L WHERE
"SF2K900001
DDNAME BETWEEN TABLEFR AND TABLETO.
"SF2K900001
  CHECK D030L-DDMODIFY NE USR.
"SF2K900001
  SELECT * FROM D030T WHERE
"SF2K900001
    DDLANGUAGE EQ SY-LANGU AND
"SF2K900001
    DDNAME EQ D030L-DDNAME.
"SF2K900001
  ENDSELECT.
"SF2K900001
  EXTRACT HEADER.
"SF2K900001
ENDSELECT.
"SF2K900001
```

Validation Rules

Because of the nature of the impact of changes in validation rules to the entire SAP system operation, any developer who is considering altering a validation rule must send a message to MIT-ABAP@MITVMA.MIT.EDU as early in the development process as possible. At a minimum this notification must be sent one week before any development is transported to the test and integration system (SF5).

Configuration Changes that alter Transaction Fields

As with changes to validation rules, any configuration changes that alter any transaction field, must be submitted in a mail message to MIT-ABAP@MITVMA.MIT.EDU as early as possible with a minimum of one week notification before transport to the test and integration system. Configuration changes that alter the "required", "optional", or "not permitted" status of fields is included in this specification.

V. PERFORMANCE STANDARDS

General Performance Standards

1.	"Dead" code
	Avoid leaving "dead" code in the program. Comment out variables that are not referenced and code that is not executed. To analyze the program, use the Program Analysis function in SE38 -> Utilities -> Program Analysis.
2.	**Use logical databases**
	Choose the most efficient logical data base possible. Study the selection criteria and which secondary indexes are used for that view. Provide the appropriate selection criteria to limit the number of data base reads. Force users to provide selection criteria by evaluating the selection criteria entered on the selection screen during the AT SELECTION-SCREEN event. Finally, when possible take advantage of the matchcodes to increase speed.
3.	**Subroutine usage**
	For good modularization, the decision of whether or not to execute a subroutine should be made before the subroutine is called. For example:

This is better:

```
IF f1 NE 0.
  PERFORM sub1.
ENDIF.
FORM sub1.
  ...
ENDFORM.
```

Than this:

```
PERFORM sub1.
  FORM sub1.
IF f1 NE 0.
  ...
ENDIF.
ENDFORM.
```

4.	**IF statements**
	When coding IF tests, nest the testing conditions so that the outer conditions are those which are most likely to fail. For logical expressions with AND, place the mostly likely false first and for the OR, place the mostly likely true first.
5.	**CASE vs. nested IFs**
	When testing fields "equal to" something, one can use either the nested IF or the CASE statement. The CASE is better for two reasons. It is easier to read and after about five nested IFs the performance of the CASE is more efficient.
6.	**MOVE-ing structures**
	When records a and b have the exact same structure, it is more efficient to MOVE a TO b than

to MOVE-CORRESPONDING a TO b.

```
MOVE BSEG TO
*BSEG.
```

is better than

```
MOVE-
CORRESPONDING
BSEG TO *BSEG.
```

7. SELECT and SELECT SINGLE

When using the SELECT statement, study the key and always provide as much of the left-most part of the key as possible. If the entire key can be qualified, code a SELECT SINGLE not just a SELECT. If you are only interested in the first row or there is only one row to be returned, using SELECT SINGLE can increase performance by up to 3x.

8. Small internal tables vs. complete internal tables

In general it is better to minimize the number of fields declared in an internal table. While it may be convenient to declare an internal table using the LIKE command, in most cases, programs will not use all fields in the SAP standard table. For example:

Instead of this:

```
data:  tvbak like
vbak occurs 0
with header line.
```

Use this:

```
data: begin of
tvbak occurs 0,
        vbeln
like vbak-vbeln,
         ...
      end of
tvbak.
```

9. Row-level processing of a table

Selecting data into an internal table using an array fetch versus a SELECT-ENDELECT loop will give at least a 2x performance improvement. After the data has been put into the internal data, then row-level processing can be done. For example, use:

```
select ... from
table <..>
           into
<itab>
(corresponding
fields of itab)
           where
...

loop at <itab>
  <do the row-
level processing
here>
endloop.
```

instead of using:

```
select ... from
table <..>
           where
...
  <row-level
processing>
  append <itab>.
endselect.
```

10.	**Row-level processing and SELECT SINGLE**
	Similar to the processing of a SELECT-ENDSELECT loop, when calling multiple SELECT-SINGLE commands on a non-buffered table (check Data Dictionary -> Technical Info), you should do the following to improve preformance: Use the SELECT into <itab> to buffer the necessary rows in an internal table, next sort the rows by the key fields, then use a READ TABLE WITH KEY ... BINARY SEARCH in place of the SELECT SINGLE command. Note that this only make sense when the table you are buffering is not too large (this must be made on a case by case decision basis).
11.	**READing single records of internal tables**
	When reading a single record in an internal table, the READ TABLE WITH KEY is not a direct READ. Therefore, SORT the table and use READ TABLE WITH KEY BINARY SEARCH.
12.	**SORTing internal tables**
	When SORTing internal tables,

specify the fields to SORTed.

```
SORT ITAB BY FLD1 FLD2.
```

is more efficient than

```
SORT ITAB.
```

13.	Number of entries in an internal table

To find out how many entries are in an internal table use DESCRIBE.

```
DESCRIBE TABLE ITAB LINES CNTLNS.
```

is more efficient than

```
LOOP AT ITAB.
  CNTLNS = CNTLNS + 1.
ENDLOOP.
```

14.	Length of a field

To find out the length of a field use the string length function.

```
FLDLEN = STRLEN (FLD).
```

is more efficient than

```
IF FLD CP '* #'.
ENDIF.
FLDLEN = SY-FDPOS.
```

15.	Performance diagnosis

To diagnose performance problems, it is recommended to use the SAP transaction SE30, ABAP/4 Runtime Analysis. The utility allows statistical analysis of

transactions and programs.

16.	Nested SELECTs versus table views
	Since OPEN SQL does not allow table joins, often a nested SELECT loop will be used to accomplish the same concept. However, the performance of nested SELECT loops is very poor in comparison to a join. Hence, to improve performance by a factor of 25x and reduce network load, you should create a view in the data dictionary then use this view to select data.
17.	**If nested SELECTs must be used**
	As mentioned previously, performance can be dramatically improved by using views instead of nested SELECTs, however, if this is not possible, then the following example of using an internal table in a nested SELECT can also improve performance by a factor of 5x: Use this: **form select_good.** **data: tvbak like vbak occurs 0 with header line.** **data: tvbap like vbap occurs**

```
0 with header
line.

  select * from
vbak into table
tvbak up to 200
rows.
  select * from
vbap
           for
all entries in
tvbak
           where
vbeln=tvbak-
vbeln.
  endselect.
endform.
```

Instead of this:

```
form select_bad.
  select * from
vbak up to 200
rows.
    select * from
vbap where vbeln
= vbak-vbeln.
    endselect.
  endselect.
endform.
```

18. SELECT * versus SELECTing individual fields

In general, use a SELECT statement specifying a list of fields instead of a SELECT * to reduce network traffic and improve performance. For tables with only a few fields the improvements may be minor, but many SAP tables contain more than 50 fields when the program needs only a few. In the latter case, the performace gains can be substantial. For example:

Use:

```
select vbeln
auart vbtyp from
table vbak
  into (vbak-
vbeln, vbak-
auart, vbak-
vbtyp)
  where ...
```

Instead of using:

```
select * from
vbak where ...
```

19.	**Avoid unnecessary statements**

There are a few cases where one command is better than two. For example:

Use:

```
append <tab_wa>
to <tab>.
```

Instead of:

```
<tab> = <tab_wa>.
append <tab>
(modify <tab>).
```

And also, use:

```
if not <tab>[] is
initial.
```

Instead of:

```
describe table
<tab> lines
<line_counter>.
if <line_counter>
> 0.
```

20.	**Copying or appending internal tables**

Use this:

```
<tab2>[] =
<tab1>[].  (if
<tab2> is empty)
```

Instead of this:

```
loop at <tab1>.
  append <tab1>
to <tab2>.
endloop.
```

However, if <tab2> is not empty and should not be overwritten, then use:

```
append lines of
<tab1> [from
index1] [to
index2] to
<tab2>.
```

ABAP/4 Tuning Checklist

The general performance standards above outline ways to increase efficiency from many different perspectives, the following checklist was developed by SAP to quickly review the most common performance problems. Please note that some of the information may overlap with the general performance section.

Is the program using SELECT * statements?

Convert them to SELECT column1 column2 or use projection views.

Are CHECK statements for table fields embedded in a SELECT ... ENDSELECT loop?

Incorporate the CHECK statements into the WHERE clause of the SELECT statement.

Do SELECTS on non-key fields use an appropriate DB index or is the table buffered?

Create an index for the table in the data dictionary or buffer tables if they are read only or read mostly.

Is the program using nested SELECTs to retrieve data?

Convert nested SELECTs to database views, DB joins (v4.0), or SELECT xxx FOR ALL ENTRIES IN ITAB.

Are there SELECTs without WHERE condition against files that grow constantly (BSEG, MKPF, VBAK)?

Program design is wrong - back to the drawing board.

Are SELECT accesses to master data files buffered (no duplicate accesses with the same key)?

Buffer accesses to master data files by storing the data in an internal table and filling the table with the READ TABLE ... BINARY SEARCH method

Is the program using SELECT ... APPEND ITAB ... ENDSELECT techniques to fill internal tables?

Change the processing to read the data immediately into an internal table (SELECT VBELN AUART ... INTO TABLE IVBAK ...)

Is the program using SELECT ORDER BY statements?

Data should be read into an internal table first and then sorted unless there is an appropriate index on the ORDER BY fields

Is the programming doing calculations or summartions that can be done on the database via SUM, AVG, MIN, or MAX functions of the SELECT statement?

Use the calculation capabilities of the database via SELECT SUM, ...

Are internal tables processed using the READ TABLE itab WITH KEY ... BINARY SEARCH technique?

Change table accesses to use BINARY SEARCH method

Is the program inserting, updating, or deleting data in dialog mode (not via an update function module)?

Make sure that the program issues COMMIT WORK statements when one or more logical units of work (LUWs) have been processed.

VI. NAMING STANDARDS

In general, any custom developed programs or objects should be named with the 'Z' for the first character. SAP has designated objects beginning with the letters 'Y' and 'Z' as customer named objects and insures that this name space will not be overwritten during an upgrade.

Also, SAP regularly updates the recommended customer name ranges for all development objects. For objects not included in this guide, please consult this SAP-created document. It is located on the SAP OSS system in note #16466.

ABAP Reports

ABAP report names consist of 8 characters, and should be formatted as follows:

Position	Usage
1	'Y' if program will not be migrated to production
	'Z' if program will be migrated to production
2-3	Application Name Abbreviation (see table following)
4-7	Any 4-character acronym describing the program

Examples

ZARI007 Cashier Feed Validation Request

ZCOR001 Report CO objects missing funds and/or fund centers

However, if the ABAP program is being created as a global data include or subroutine include, it should be formatted as follows:

Position	Usage
1	'Z' as required for customer development
2-5	same as positions 4-7 in the main program
6-8	'TOP' if used for global data, data declarations, table declarations
	'F##' for subroutines where ## is a number, 00 through 99.

Examples

ZFCCMTOP Credit card TOP Include

ZFCCMF01 Credit card FORM Include

Classification Objects and Class Types

Since classification objects and class types are more like programming data constructs than they are like configuration data, MIT naming conventions should be followed when creating new classification objects or class types. To differentiate between SAP supplied classes and MIT developed classes the prefix 'MIT_' should be used followed by either the SAP data element or a descriptive abbreviation. For example, when creating a characteristic for a 'Profit Center' where the data element is 'PRCTR' the name would be 'MIT_PRCTR'. In addition, SAP recommends to use only letters from A-Z, numbers 0-9, and the underscore character in the name.

Development Classes

Development classes should only be created when a specific new project requires all components to be linked for organizational purposes. Before creating a new development class, please consult with the R3 Admin team (r3-admin@mit.edu).

Directories for file I/O

Most SAP system interfaces using flat ASCII files for input or output should use the following directory paths for temporary and permanent interface files.

To R/3 from an EXTERNAL SYSTEM:
/usr/bridges/[environment name]/sapin/

To an EXTERNAL SYSTEM from R/3:
/usr/bridges/[environment name]/sapout/

Files written by one ABAP program and read by another:

```
/usr/bridges/[environment name]/saptmp/

Files already used by R/3:
/usr/bridges/[environment name]/archive/
```

The variable [environment name] should be determined at run-time by the program. To generate this name, use the function Z_DETERMINE_FUNC_AREA. This function will identify the [environment name] section of the path based on the system id and the client of the calling R/3 system.

Sample code follows:

```
data: begin of myfile,
        part1(13) value '/usr/bridges/',
        part2(15), "dev2 or tst1 or prod
        part3(9) value '/saptmp/',
        part4(30) value 'flname',
      end of myfile.
…
initialization.
  call function 'Z_DETERMINE_FUNC_AREA'
    importing
      func = myfile-part2
    exceptions
      unknown_sysid = 1
      unknown_mandt = 2.
  if sy-subrc eq 0.
    condense myfile no-gaps.
  endif.
```

Function Groups

Since there is a limited amount of logical naming space available for function groups, generally new groups should only be created when it is not possible to add functions to existing groups.

Function Groups are named using the following convention:

Position	Usage
1	'Z' as required for customer development
2-3	Application Name Abbreviation (see Appendix B)
4	Sequential group number

For example, ZBR0 is the function group for application "BR" for "Bridges" and number '0' sequentially. Before creating a new function group, please consult with David Rosenberg (rosenberg@mit.edu).

Function Modules

Function module names should be as descriptive as possible given that there are 30 characters available for naming. They should be formatted as follows:

Position	Usage
1	'Z' as required for customer development
2	'_' (an underscore)
3-4	Application Name Abbreviation (see Appendix B)
5-30	any use of descriptive words separated by underscores

Module Pools

Module pools should be named as follows:

Position	Usage
1-3	'SAPM' as required by the system to denote online modules
5	'Z' to denote customer named object
6-7	Application Name Abbreviation (see Appendix B)
8	Sequential Number (0-9)

Module pool includes should be named as follows:

Position	Usage
1	'M' as required by the system to denote online modules
2	'Z' to denote customer named object
3-4	Application Name Abbreviation (see Appendix B)
5	Sequential Number (0-9) corresponding to main program
6	'I' for PAI modules, 'O' for PBO modules, 'F' for subroutines
7-8	Sequential Number (00-99)
6-8	'TOP' if the include is the global data include

Requests & Tasks

VII. TIPS & TRICKS

ABAP: Passing unknown table structures into functions

Typically function modules must specify the exact structure of each parameter they are passed, hence runtime structures cannot be handled easily in normal function module creation. However, by using the ABAP programming construct ASSIGN COMPONENT idx OF STRUCTURE record to <FS>, a function can determine the structure of a record. For an example of how to develop a function to do this refer to function Z_LIST_SELECT_OPTIONS which reads the structure of a RANGES table in a SELECT-OPTION parameter.

ABAP: Multi-dimensional arrays

Support for one and two-dimensional arrays within ABAP is provided through the use of internal tables. However, in the case of a 3-dimensional array, ABAP has no naitive programming constucts. While a method could be devised to index a 3-dimensional array via internal tables, the following code fragment demonstrates an alternative approach using field symbols as pointers into data structures:

```
* This code fragment illustrates the use of a field symbol and
an
* ASSIGN statement to get the effect of having multi-
dimensional
* subscripted arrays. There are several costs of this method.
One is
* performance and the other is that EVERY cell in the multi-
dimensional
* array must be declared in a DATA (or CONSTANT, etc.)
statement at the
* beginning of the program.

* Using this method, you would have to have DATA statements
like the
* following four for EACH CELL of the multi-dimensional array.

DATA: VARA_001_005_006(11) VALUE 'Some value!'.
DATA: VARA_002_005_006(11) VALUE 'Other value'.
DATA: VARA_003_005_006(11) VALUE 'Next value!'.
DATA: VARA_004_005_006(11) VALUE 'The target!'.

DATA: BEGIN OF POINTER,
STEM(5) TYPE C VALUE 'vara_',
PART1(3) TYPE N,
SEP1(1) TYPE C VALUE '_',
PART2(3) TYPE N,
SEP2(1) TYPE C VALUE '_',
PART3(3) TYPE N,
END OF POINTER.

DATA POINTER_DEFAULT(11) VALUE 'The default'.

FIELD-SYMBOLS: <INDIRECT>.

* This is the example of using the multi-dimensional array in
the body
```

```
* of a program. You would have to have use something like the
following
* PERFORM for each change to the subscripts in a reference to
the
* multi-subscripted variable.

PERFORM ASSIGN_POINTER_3 USING 'vara_' 4 5 6.
WRITE: / 'Indirectly referencing through pointer, we get:',
<INDIRECT>.

PERFORM ASSIGN_POINTER_3 USING 'vara_' 10 11 12.
WRITE: / 'Indirectly referencing through pointer, we get:',
<INDIRECT>.

PERFORM ASSIGN_POINTER_3 USING 'vara_' 1 5 6.
WRITE: / 'Indirectly referencing through pointer, we get:',
<INDIRECT>.

FORM ASSIGN_POINTER_3 USING STEM SUB1 SUB2 SUB3 .

POINTER-STEM = STEM.
POINTER-PART1 = SUB1.
POINTER-PART2 = SUB2.
POINTER-PART3 = SUB3.

ASSIGN (POINTER) TO <INDIRECT>.
IF SY-SUBRC <> 0.
ASSIGN POINTER_DEFAULT TO <INDIRECT>.
ENDIF.

ENDFORM. " ASSIGN_POINTER_3
```

ABAP: Type coercion

Type coercion is typically defined as the means to force a variable of one type to be another type. In ABAP programming, type coercion is most useful in ABAP for the ASCII conversion of character data. The following code fragment is an example of coercion which outputs ASCII character data:

```
data: begin of hex,
        byte type x,
      end of hex.
data: char type c.
data: temp like sy-index.

do 224 times.
  temp = sy-index + 31.
  hex-byte = temp.
  char = hex.
  write: / 'Decimal =', temp.
  write:   'Hex     =', hex-byte.
  write:   'View    =', char.
enddo.
```

BDC: Formatting date fields to be compatible with user preferences

When writing BDCs that read dates to be entered into SAP transactions problems occur when determining what date format to use when entering the data into its corresponding field. Assume that the ABAP program that must call the transaction

can read the 'date' field properly from the input file, but it does not know which format the data entry screen of the transaction requires as this may change based on the user's setup defaults.

The current best approach to solving this problem is:

1. Declare a date variable in the interface program to be LIKE SY-DATUM.
2. Then, store the date (read from the input file) in the date variable in the correct internal form (YYYYMMDD).
3. Write that date variable to a character string variable.
4. Use the character string variable as the value of the date field in the transaction screen.

Conversions: Legacy-to-SAP cost objects and G/L accounts

Many programs and interfaces will need to convert MIT legacy cost objects and accounts to the corresponding SAP account or object. Two custom MIT function modules have been developed for use in this conversion process.

Z_GET_COST_OBJECT

This function will accept a 5-digit account and return the related SAP 7-digit cost object. Also, by passing a 7-digit SAP cost object into the function, it will verify the existence of the cost object.

Z_GET_GL_ACCOUNT

This function will accept a 3-digit object code as input and return the related SAP 6-digit G/L account. Also, by passing a 6-digit G/L account into the function, it will verify the existence of the account.

DDIC : Creating tables, data elements and domains

[This section pending input from MIT Data Administration group]

UNIX : Calling UNIX commands and scripts from ABAP Programs

Often it is necessary to execute an external program as part of an ABAP program, however, you should be very careful in doing so because any external command is executed as the UNIX user <SID>adm. For example, if a program is written to call the command 'rm' on the development system it is executed at the system level as the user 'sf2adm'.

For an example of how to call UNIX system commands, look at program ZUNIXCMD on system SF2. Remember, you can call external programs either synchronously so that your ABAP program will wait until they complete before continuing, or you can call them asynchronously so that your ABAP program will continue execution immediately.

Earlier programs written at MIT used the 'CALL SYSTEM' command. In the future, this command should not be included in any new programs and should be removed from existing programs when they are revised.

APPENDIX A

Basic ABAP/4 List Report

```
REPORT ZSKELREP.
*---------------------------------------------------------------------
--
* Purpose:
*
*
*
*---------------------------------------------------------------------
--
* Author:
* Date:
*---------------------------------------------------------------------
--
* Modification Log:
*
*---------------------------------------------------------------------
--
TABLES: ...
DATA: ...

SELECT-OPTIONS: ...
PARAMETERS: ...

FIELD-GROUPS: HEADER, ...
FIELD-SYMBOLS: ...
* Event which occurs before the selection screen is shown to
* the user.
INITIALIZATION.

* Event which occurs each time the user hits enter on the
* selection screen. This event is ignored in background
processing.
AT SELECTION-SCREEN.

TOP-OF-PAGE.

END-OF-PAGE.

START-OF-SELECTION.

* Definition of fields for FIELD-GROUP extract
INSERT: ... INTO HEADER.

GET ...

END-OF-SELECTION.

SORT ...
LOOP ...
  AT FIRST.
  ENDAT.
  AT NEW ...
```

```
  ENDAT.
  AT END OF ...
  ENDAT.
  AT LAST.
  ENDAT.
ENDLOOP.

* Subroutines
*---------------------------------------------------------------------
--
*
*
*---------------------------------------------------------------------
--
FORM ...


ENDFORM.
```

Interactive ABAP/4 List Report

```
REPORT ZSKELINT.
*---------------------------------------------------------------------
--
* Purpose:
*
*
*
*---------------------------------------------------------------------
--
* Author:
* Date:
*---------------------------------------------------------------------
--
* Modification Log:
*
*---------------------------------------------------------------------
--
TABLES: ...
DATA: ...

SELECT-OPTIONS: ...
PARAMETERS: ...

FIELD-SYMBOLS: ...
FIELD-GROUPS: ...
* Event which occurs before the selection screen is shown to
* the user.
INITIALIZATION.

* Event which occurs each time the user presses F2 or double-
clicks
* on the line in the selection screen. This event is ignored in
* background processing.
AT SELECTION-SCREEN.

TOP-OF-PAGE.

* Top of page for sublists
TOP-OF-PAGE DURING LINE-SELECTION.
```

```
END-OF-PAGE.

START-OF-SELECTION.
GET ...

END-OF-SELECTION.

* Produce main list report SY-LSIND = 0.
SORT ...
LOOP...
WRITE:/ ...

* Hide specific fields which are if importance to the line
HIDE: ...
ENDLOOP.

* Event which occurs when user hits a particular PF key, i.e.
PF6
* The hide area and SY-LISEL are automatically available.
* Produces a sublist SY-LSIND = 1-9. PF3 is automatic, will
always
* take the user back one list level, (SY-LSIND - 1).
AT PF...

* Event which occurs when a user types =LIST in the OK code
* The hide area and SY-LISEL are automatically available.
* Produces a sublist SY-LSIND = 1-9. PF3 is automatic, will
always
* take the user back one list level, (SY-LSIND - 1).
AT USER-COMMAND.
CASE SY-UCOMM.
  WHEN 'LIST'. .....
ENDCASE.

* Event which occurs when the user places the cursor to a
specific
* line on the report and hits enter.
* The hide area and SY-LISEL are automatically available.
* Produces a sublist SY-LSIND = 1-9. PF3 is automatic, will
always
* take the user back one list level, (SY-LSIND - 1).
AT LINE-SELECTION.

* Subroutines
*---------------------------------------------------------------
--
*
*
*---------------------------------------------------------------
--
FORM ...
ENDFORM.
```

Create a Sequential Dataset

```
REPORT ZSKELOUT.
*---------------------------------------------------------------
--
* Purpose:
*
*
```

```
*
*---------------------------------------------------------------------
--
* Author:
* Date:
*---------------------------------------------------------------------
--
* Modification Log:
*
*---------------------------------------------------------------------
--
TABLES: ...
DATA: ...

SELECT-OPTIONS: ...
PARAMETERS: ...

FIELD-SYMBOLS: ...

START-OF-SELECTION.

GET ...
MOVE ... TO ...
WRITE ... TO ...
UNPACK ... TO ...
TRANSFER ... TO ...

END-OF-SELECTION.

* Subroutines
*---------------------------------------------------------------------
--
*
*
*---------------------------------------------------------------------
--
FORM ...

ENDFORM.
```

Read Sequential Dataset and Create a BDC Session

```
report zskelbdc.
*---------------------------------------------------------------------
---
* Purpose:
*
*
*
*---------------------------------------------------------------------
---
* Author:
* Date:
*---------------------------------------------------------------------
---
* Modification Log:
*
*---------------------------------------------------------------------
---
tables: ...
```

```
data: bdcdata like bdcdata occurs 0 with header line.
data: messtab like bdcmsgcoll occurs 0 with header line.
data: begin of tab occurs ...
      end of tab.

select-options: ...

parameters: ...

field-symbols: ...

*---------------------------------------------------------------
--*
start-of-selection.
  open dataset p_dataset in text mode.
  if sy-subrc <> 0.
    write: / text-e00, sy-subrc.
    stop.
  endif.

*--- Open the BDC Session -------------------------------------
--*
write: /(20) 'Create group'(i01), group.
skip.
call function 'BDC_OPEN_GROUP'
     exporting client = sy-mandt
               group = group
               user = user
               keep = keep
               holddate = holddate.
write: /(30) 'BDC_OPEN_GROUP'(i02), (12) 'returncode:'(i05),
      sy-subrc.

*----------- DYNPRO nnn -----------------------------------------
--*
perform bdc_dynpro using 'SAPMxxxx' 'nnn'.
perform bdc_field using 'TABL-FIELD' 'LITERAL'.
*----------- DYNPRO nnn -----------------------------------------
--*
perform bdc_dynpro using 'SAPMxxxx' 'nnn'.
perform bdc_field using 'TABL-FIELD' TAB-VAR.
*---------------------------------------------------------------
--*
call function 'BDC_INSERT'
     exporting tcode = tcode
     tables dynprotab = bdcdata.
write: /(25) 'BDC_INSERT'(i03), tcode,
        (12) 'returncode:'(i05), sy-subrc.
close dataset p_dataset.
*----- Close the BDC Session --------------------------------
-
call function 'BDC_CLOSE_GROUP'.
write: /(30) 'BDC_CLOSE_GROUP'(i04),
        (12) 'returncode:'(i05), sy-subrc.
end-of-selection.
*---------------------------------------------------------------
--
* Subroutines *
*---------------------------------------------------------------
--
```

```
*--- Add a line to the DYNPRO Table -------------------------
-
form bdc_dynpro using program dynpro.
  clear bdcdata.
  bdcdata-program = program.
  bdcdata-dynpro = dynpro.
  bdcdata-dynbegin = 'X'.
  append bdcdata.
endform.
*--- Add a field to a DYNPRO --------------------------------
--
form bdc_field using fnam fval.
  clear bdcdata.
  bdcdata-fnam = fnam.
  bdcdata-fval = fval.
  append bdcdata.
endform.
```

CALL TRANSACTION USING Technique

```
REPORT ZSKELCLT.
*---------------------------------------------------------------
---
* Purpose:
*
*
*---------------------------------------------------------------
---
* Author:
* Date:
*---------------------------------------------------------------
---
* Modification Log:
*
*---------------------------------------------------------------
---
tables: indx, ...

data: return_code like sy-subrc,
      message_text(120).
data: begin of trantab occurs 10.
        include structure bdcdata.
data: end of trantab.

select-options: ...
parameters: ...
field-symbols: ...

start-of-selection.
*----------- DYNPRO nnn ---------------------------------------
--*
perform bdc_dynpro using 'SAPMxxxx' 'nnn'.
perform bdc_field using 'TABL-FIELD' 'LITERAL'.
*----------- DYNPRO nnn ---------------------------------------
--*
perform bdc_dynpro using 'SAPMxxxx' 'nnn'.
perform bdc_field using 'TABL-FIELD' TAB-VAR.
*---------------------------------------------------------------
--*
call transaction ' ' using trantab mode 'N' update 'S'.
```

```
* Message handling
return_code = sy-subrc.

* The following function module is a custom routine and can
* be obtained from SAP America.
call function 'ZZ_FORMAT_MESSAGE'
     exporting message_id = sy-msgid
               message_number = sy-msgno
               var1 = sy-msgv1
               var2 = sy-msgv2
               var3 = sy-msgv3
               var4 = sy-msgv4
     importing message_text = message_text
     exceptions not_found = 4.
if sy-subrc = 4.
  message_text = 'No message text found in T100'.
endif.
if return_code = 0.
* At this point, various things can be done to make the
* process slicker.
* Send the confirmation or error to the other program via RFC.
* Store key values and confirmation document number or error
* message in an ATAB table for future processing.
  move 'Transaction posted' TO ...
  move message_text to ...
  modify ...
else.
  move 'Transaction failed' to ...
  move message_text to ...
  modify ...
* In the event of errored transactions:
* Store the internal table in the INDX database for future
online
* processing of the SAP transaction.
  export trantab to indx(..) id ...
endif.
* Or create a batch input session for future processing.
refresh trantab.
end-of-selection.
*---------------------------------------------------------------
--
* Subroutines *
*---------------------------------------------------------------
--
*--- Add a line to the DYNPRO Table --------------------------
-
form bdc_dynpro using program dynpro.
  clear bdcdata.
  bdcdata-program = program.
  bdcdata-dynpro = dynpro.
  bdcdata-dynbegin = 'X'.
  append bdcdata.
endform.
*--- Add a field to a DYNPRO ---------------------------------
--
form bdc_field using fnam fval.
  clear bdcdata.
  bdcdata-fnam = fnam.
  bdcdata-fval = fval.
  append bdcdata.
endform.
```

Appendix B: MIT Application Names Abbreviations

The following two-character abbreviations are used to identify certain projects or modules when naming programming objects.

2-Character Abbr.	Application Name
AR	Accounts Receivable
AU	Authorization Checks
BR	Bridges
BP	Buy-Pay Process (incl. Purchasing)
CD	Change Documents
CO	Controlling
FC	Credit Card
GL	Finance Functions
L*	Labor Distribution System
PM	Plant Maintenance
R3	Basis
RQ	Requisitions
TS	Time Sheet (CATS Related)
WF	Workflow
WH	Warehouse