TEMENOS™

TEMENOS

Programming Standards

User Guide

Table of Contents

Overview

This chapter defines the standards that must be adhered to when writing programs that operate in the T24 environment. The rules are designed to make your code legible, maintainable and operational within both the T24 and the T24 Graphical User environments.

The structure of T24 is designed around the use of business objects; hence unless these rules are strictly adhered to your code will not operate correctly if the functionality of the object is increased. T24 is a constantly changing product; therefore your edited piece of code written for use at one release may be completely useless by the next release.

Adherence to the standards means that other people experienced in T24 programming can maintain your code. Even if you feel that your particular style is superior, remember that the standards allow other people to support your programs.

Why coding conventions?

The main reason for using a consistent set of coding conventions is to standardise the structure and coding style of the T24 applications so that you and others can easily read and understand the code.

Good coding conventions result in precise, readable and unambiguous source code that is consistent with other language conventions and as intuitive as possible.

This set of coding conventions should define the minimum requirements necessary to accomplish the purposes discussed above, leaving the programmer free to create the programs logic and functional flow.

The object is to make the program easy to read and understand without cramping the programmer's natural creativity with excessive constraints and arbitrary restrictions.

To this end, the conventions suggested in this chapter are brief and suggestive. They do not cover every eventuality, not do they specify every type of informational comment that could be valuable.

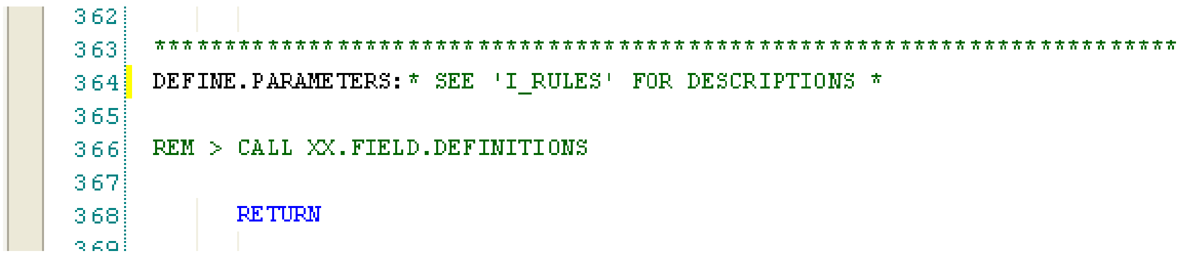
There are by the very nature of a product that has been developed over many years examples of code that does not conform to the standards required, however all new code should follows the standards laid out in this document. Where feasible when amending programs that do not comply with the standards efforts should be made to standardise the program.

Program Structure

- All routines in T24 are called subroutines. Routines must not be written in such a way that they are executed directly from jSHELL.

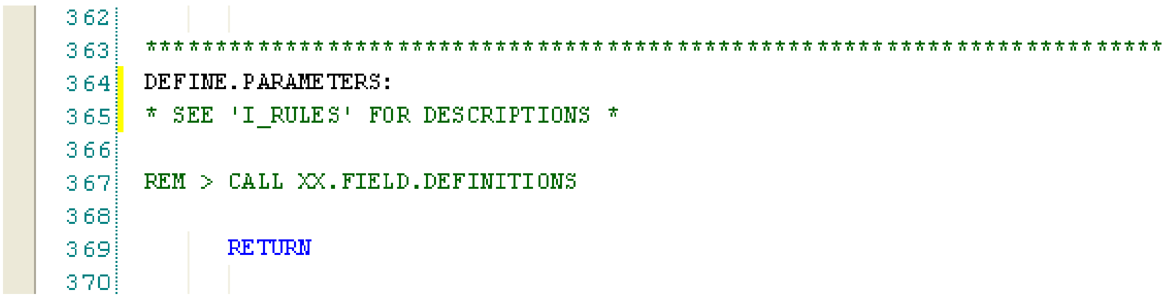
- TEMPLATE programs should be less than 2000 lines. New template programs should contain very little code, all code should be called from the subroutines defined in the template e.g. FIELDS, CHECK.FIELDS & CROSSVAL.
- Subroutines should be less than 600 lines. Ideally the code should be broken down into internal subroutines that can be viewed in its entirety on one screen (i.e. 80 character wide by 23 lines deep).
- **Do not** write TOP DOWN code. Each routine should have one main controlling section with the detailed code written as subroutines. **Do not** assume field 1 will be input before field 2; remember client versions.
- All code should be structured in a modular fashion and broken into small units/paragraphs (less than 100 lines) and each unit should have a single entry and exit point.
- Avoid deeply nested IF's and large case constructs. If an IF construct exceeds 20 lines then use a GOSUB - nothing is harder to follow in a program than an IF that starts on page 1 and ends on page 21.
- Labels, Variables and Routines should not use the same names.
- Labels and variables should have meaningful names. Numeric labels should not be used. Labels must exist on their own line with no other text appended to the label.

For example, the following is not permitted



```
*************************************************************************
DEFINE.PARAMETERS:* SEE 'I_RULES' FOR DESCRIPTIONS *

REM > CALL XX.FIELD.DEFINITIONS

      RETURN
```

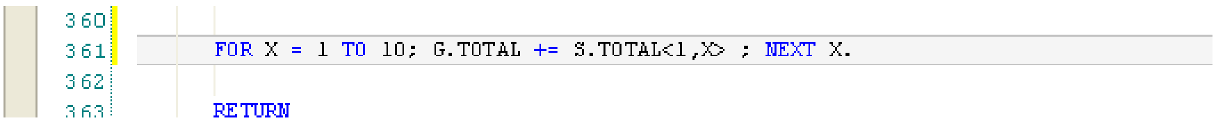
The following should be used instead



```
*************************************************************************
DEFINE.PARAMETERS:
* SEE 'I_RULES' FOR DESCRIPTIONS *

REM > CALL XX.FIELD.DEFINITIONS

      RETURN
```

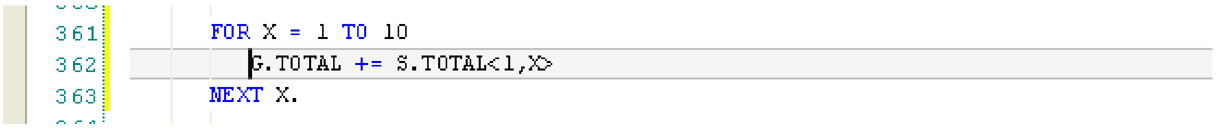
- Do not use multi-statement lines:

For example, do not use the following



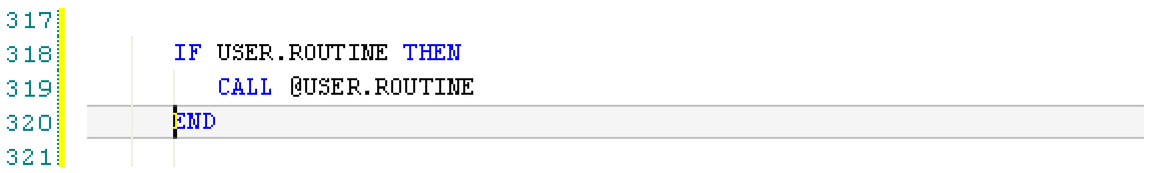
```
    FOR X = 1 TO 10; G.TOTAL += S.TOTAL<1,X> ; NEXT X.

    RETURN
```

Use the following instead,



```
    FOR X = 1 TO 10
       G.TOTAL += S.TOTAL<1,X>
    NEXT X.
```

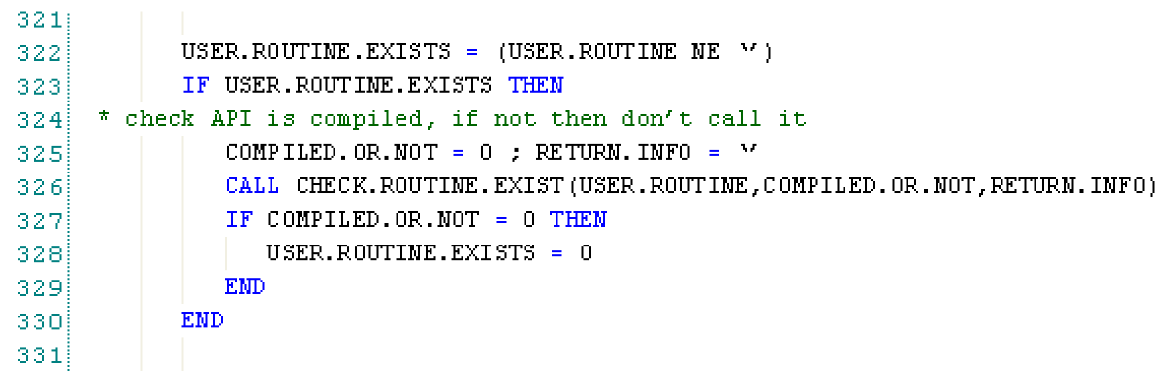
- Code should not be commented out - remove it.
- Do not use STOP, ABORT or RETURN TO.
- Do not use GOTO even if it is to exit a unit (or is used to retry the read of a locked record).
- File variable names must be F.filename (Where file name is the EXACT name of the file e.g. F.ACCOUNT); similarly record variables should be R.filename.
- Fields must never be referenced by their field number (except in conversion routines). You must use the field names supplied in the standard inserts.
- Avoid using FOR…NEXT loops, use the more efficient LOOP…REMOVE syntax.
- Do not cut and paste code, use an internal subroutine/paragraph for code that is very similar or used several times. When maintaining/enhancing older programs with duplicated code like this in the area being modified, re-write this section to remove the duplication. This will make future maintenance and enhancement easier.
- All subroutines, both the subroutine itself and internal subroutines, should return from a single exit point.
- Do not pass common variables as arguments to subroutines, e.g. ETEXT should only used as the error value returned from F.READ. The use of common variables in this way can lead to unexpected errors.
- When calling API interfaces or user defined subroutines do not use the program as the flag as this is replaced by a direct memory address for the routine once called, for example the following will fail if executed a second time as USER.ROUTINE will be memory address not a string. Additionally the API should be checked to ensure that the call will not fail, using the standard routine CHECK.ROUTINE.EXIST. This check should be performed once and once only as the check takes some time to execute, especially in jBASE.



```
    IF USER.ROUTINE THEN
       CALL @USER.ROUTINE
    END
```

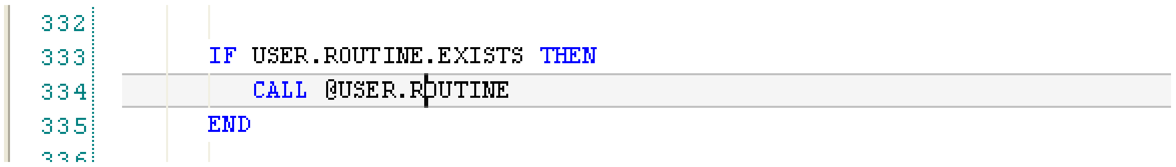
Should be replaced with a separate initialisation and call:

In program initialisation section



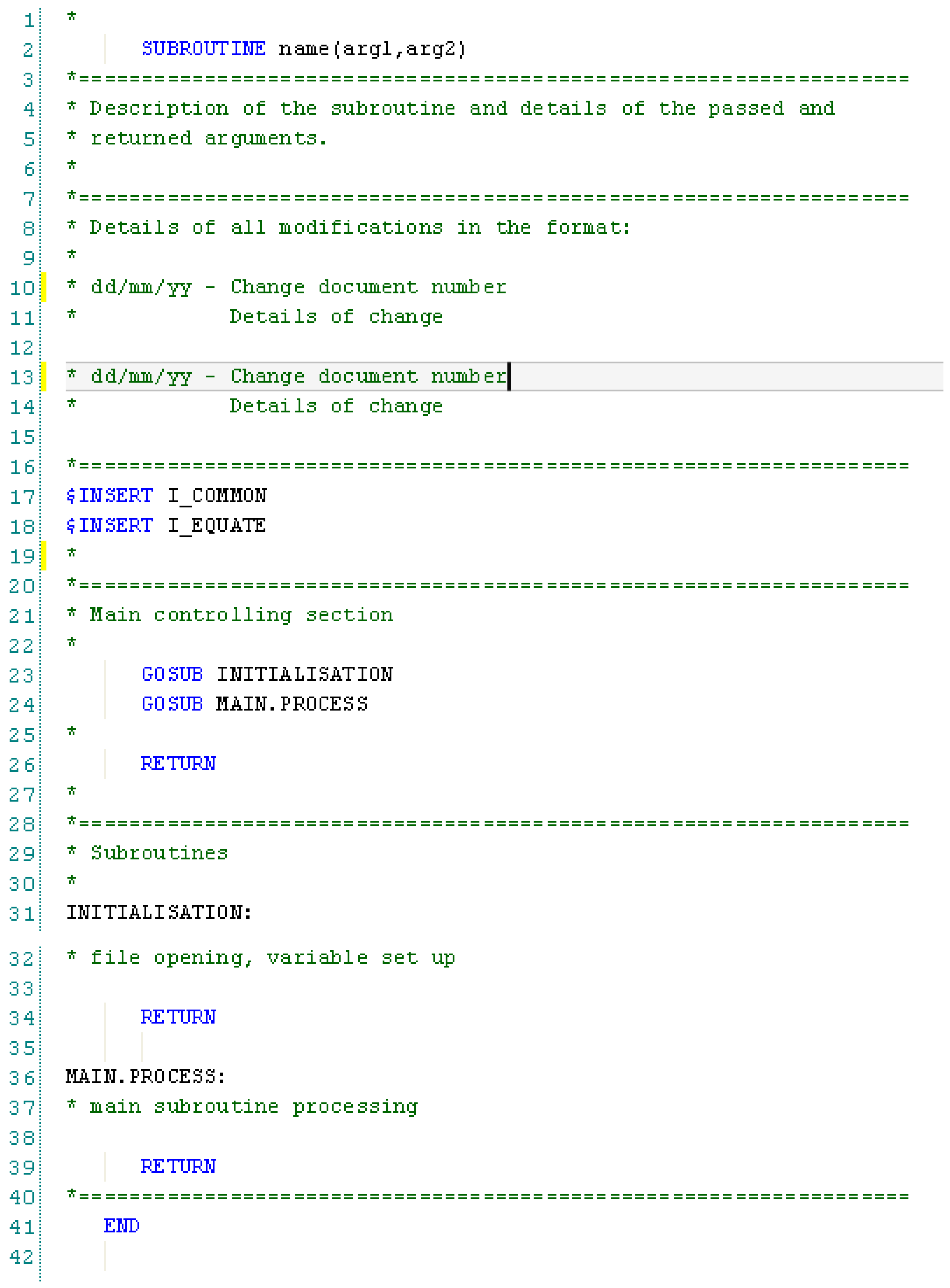
```
        USER.ROUTINE.EXISTS = (USER.ROUTINE NE '')
        IF USER.ROUTINE.EXISTS THEN
* check API is compiled, if not then don't call it
           COMPILED.OR.NOT = 0 ; RETURN.INFO = ''
           CALL CHECK.ROUTINE.EXIST(USER.ROUTINE,COMPILED.OR.NOT,RETURN.INFO)
           IF COMPILED.OR.NOT = 0 THEN
              USER.ROUTINE.EXISTS = 0
           END
        END
```

In main process section



```
        IF USER.ROUTINE.EXISTS THEN
           CALL @USER.ROUTINE
        END
```

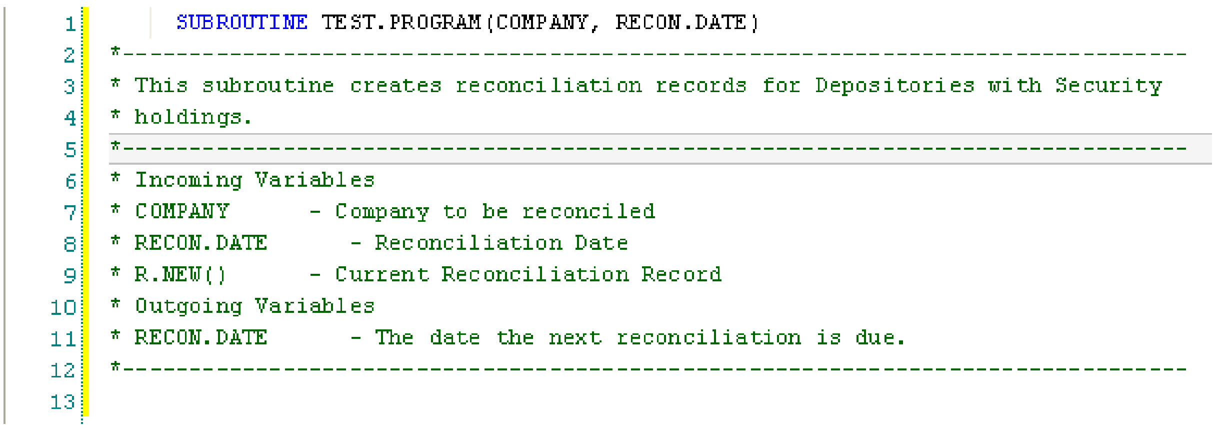
- Routines should be structured thus:



```
*
      SUBROUTINE name(arg1,arg2)
*==================================================================
* Description of the subroutine and details of the passed and
* returned arguments.
*
*==================================================================
* Details of all modifications in the format:
*
* dd/mm/yy - Change document number
*            Details of change

* dd/mm/yy - Change document number
*            Details of change

*==================================================================
$INSERT I_COMMON
$INSERT I_EQUATE
*
*==================================================================
* Main controlling section
*
      GOSUB INITIALISATION
      GOSUB MAIN.PROCESS
*
      RETURN
*
*==================================================================
* Subroutines
*
INITIALISATION:
* file opening, variable set up

      RETURN

MAIN.PROCESS:
* main subroutine processing

      RETURN
*==================================================================
   END

```

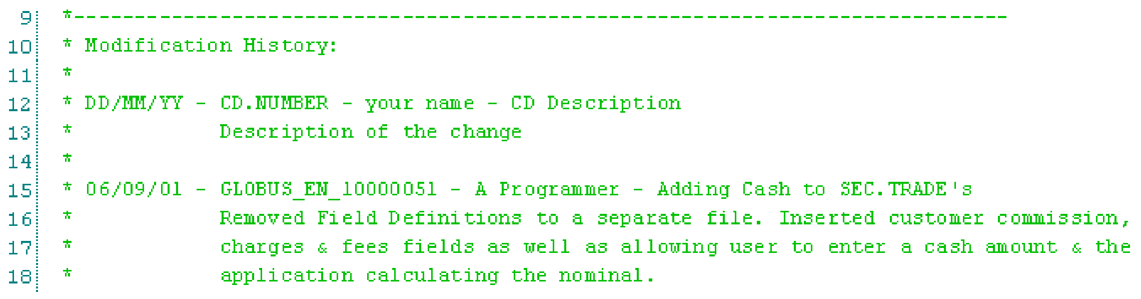
Comments in Programs

- **DO NOT** comment out existing code **DELETE** it.
- **DO NOT** use asterisks (*) to create coding breaks, either use the "*---------------",in toolbox the following button *+−, to break up sections or use blank lines to make code more readable.
- **DO NOT** embed the CD numbers in the code as separate lines, always append to a line of code.
- The first line of the program should be a comment line; the second line should be the SUBROUTINE (or PROGRAM) statement. The next few lines of the program should be a brief description of what the program does.
- All called arguments and common variables used should be documented in the called routine. Specify which are inward and which are outward.



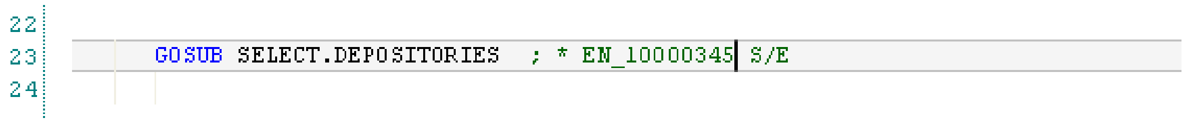
```
    SUBROUTINE TEST.PROGRAM(COMPANY, RECON.DATE)
*--------------------------------------------------------------------------------
* This subroutine creates reconciliation records for Depositories with Security
* holdings.
*--------------------------------------------------------------------------------
* Incoming Variables
* COMPANY      - Company to be reconciled
* RECON.DATE      - Reconciliation Date
* R.NEW()      - Current Reconciliation Record
* Outgoing Variables
* RECON.DATE      - The date the next reconciliation is due.
*--------------------------------------------------------------------------------
```

- Update the modifications section with the changes you have made, include the date, the full CD number, your name, CD title & how the CD changes the program being modified.



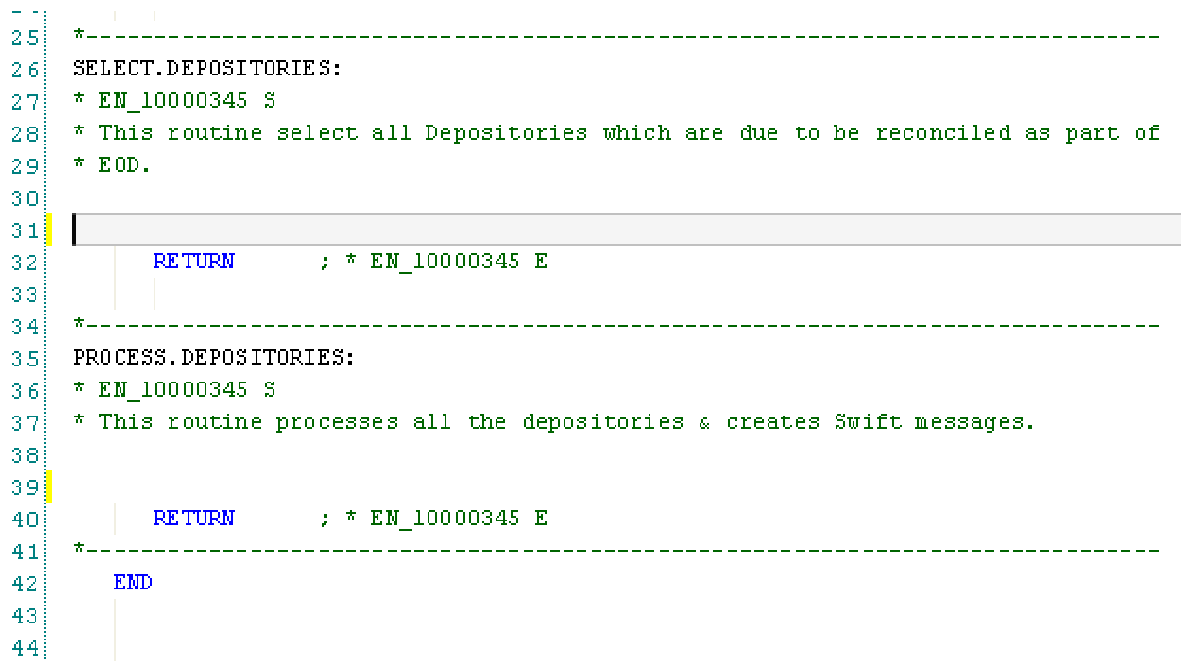
```
*----------------------------------------------------------------------------
* Modification History:
*
* DD/MM/YY - CD.NUMBER - your name - CD Description
*            Description of the change
*
* 06/09/01 - GLOBUS_EN_10000051 - A Programmer - Adding Cash to SEC.TRADE's
*            Removed Field Definitions to a separate file. Inserted customer commission,
*            charges & fees fields as well as allowing user to enter a cash amount & the
*            application calculating the nominal.
```

- Put the CD reference on the end of the code. Use S to indicate Start of change & E to indicate End of change.
- If the change only affects one line of code then use S/E.



```
    GOSUB SELECT.DEPOSITORIES   ; * EN_10000345 S/E
```

- Use the "*-----------------------------" to separate subroutines so that it is easier for other people to read the code. In toolbox the following button
- Put the CD number on the 1st label and the final RETURN so that other people can see when the subroutine was added.
- Use the first line after the label to describe what this routine does.



```
*-------------------------------------------------------------------------------
SELECT.DEPOSITORIES:
* EN_10000345 S
* This routine select all Depositories which are due to be reconciled as part of
* EOD.


      RETURN      ; * EN_10000345 E

*-------------------------------------------------------------------------------
PROCESS.DEPOSITORIES:
* EN_10000345 S
* This routine processes all the depositories & creates Swift messages.


      RETURN      ; * EN_10000345 E
*-------------------------------------------------------------------------------
   END
```

Every internal subroutine/paragraph must begin with comments explaining the purpose of that section of code. Again this should include a description of the values supplied and returned the same as the main routine.

Naming Conventions

Use I_RULES in GLOBUS.BP to learn more about the common variables used in T24.

Constants and Variables

There are three basic levels of variables in T24 programs, Global, Product and Subroutine level.

Global Variables

Global variables are defined in I_COMMON. This insert is included in every T24 program and therefore this common block is available to all programs. The prefix of C$ has been adopted for global variables, however there are many variables in I_COMMON that were defined before this standard was adopted and consequently do not use this prefix.

Extreme care should be taken if you change a global variable in your program. You are strongly advised to avoid doing so.

Product Level Variables

Product level variables are defined in the product level common block e.g. I_RP.COMMON and are available to all subroutines that include that common block. Typically all the subroutines in a product level variable are included in the common block. The prefix should be XX$ where XX is the product code, e.g. RP$ for Repo.

Subroutine Level Variables

These variables are only used in the subroutine in which they are defined. They have no prefix and can only be passed to another subroutine as an argument in the CALL statement.

Always use meaningful variable names, variable names such as X are difficult to read and convey no information to their use.

Variable Prefixes

Level	**Defined in**	**Prefix**	**Example**
Global	I_COMMON	C$	C$EB.PHANTOM.ID
Product Level	I_XX.COMMON e.g. I_RP.COMMON	XX$ e.g. RP$	RP$REPO.TYPE
Subroutine level	The subroutine	None	R.ACCOUNT

Subroutine Names and Variables

Subroutine names and variable names should be in upper case and should be as long as necessary to describe their purpose. However subroutine names are limited to 35 characters. In addition subroutine names should begin with the product prefix and then describe the routine, e.g. RP.CHECK.FIELDS.

Labels and variables should have meaningful names; numeric labels should not be used.

For frequently used or long term, standard abbreviations are recommended to help keep name lengths reasonable. In general variable names greater than 32 characters can be difficult to read. If the subroutine name exceeds 35 characters then where possible the application name should be retained and the remaining abbreviated, e.g. XX.application.FIELD.DEFINITIONS should be abbreviated to XX.application.FIELD.DEFS.

Subroutine names should not include the “$” or “_” character. This is incompatible with Oracle™.

Subroutine names cannot be named with an extension of B (i.e. SC.CALC.YIELD.B). The extension b is a reserved extension for c code and case insensitive systems such as Windows and iSeries machines treat B the same as b. To ensure that no problems are encountered, it is best to avoid using program names that end with ‘extensions’ that may be interpreted by the underlying system, i.e. Windows extensions .doc, .xls, .txt, Language extensions .c, .C, .cpp, .jar, .java, .class.

Standard Variables

Files and records

		Example
File Variable	F.filename	F.ACCOUNT
File name	FN.filename	FN.ACCOUNT
Record variable	R.filename	R.ACCOUNT

Where filename is the exact name of the file. Filename variables should only be used when required, normally a constant should be used e.g. CALL OPF(‘F.ACCOUNT’,F.ACCOUNT).

Standard Abbreviations

When using abbreviations make sure they are consistent throughout the entire application. Randomly switching between ACCR.INT and ACCRUED.INTEREST within a subroutine will lead to unnecessary confusion.

Field Names and Numbers

Fields must never be reference by their field number, except in conversion subroutines. You should always use the field name as specified in the standard inserts.

Standard Field Names

To standardise the way in which the T24 database is structured, certain types of fields have a standard definition that should be followed.

Field Type	Field name to be used
ACCOUNT field	ACCOUNT.NO
CUSTOMER field	CUSTOMER.NO
CURRENCY field	CURRENCY
Long Description	DESCRIPTION
Short Description	SHORT.DESC

Commons and Inserts

- All routines MUST have I_COMMON and I_EQUATE inserted at the start of the program.
- All common blocks must be specified in an insert (they must not be coded directly into a routine). It is much easier to maintain an insert.
- All common variables should be prefixed by C$... so that they are readily identifiable in the routine.
- Field names must be stored in their own insert as created by FILE.LAYOUT.
- Code must not be written in inserts. Write a called subroutine instead.
- Inserts must not contain inserts.
- Always add 'reserved' variables to common blocks as this avoids common mismatches when adding new variables.
- All common blocks must be 'named' commons.
- Patches may contain new or changed inserts. When Insert files are released through patches, they reside in PATCH.BP directory. If any local developments use these insert files then copy them from PATCH.BP to GLOBUS.BP directory and recompile any associated programs.

Standard Subroutines

The following routines MUST be used when performing the following operations:

Operation	Routines
Read, writes etc.	F.READ, F.WRITE ... CACHE.READ. This subroutine should only be used to read "static" data. This should be data that does not change frequently, for example either system wide or company based parameter records.
File opens	OPF
Screen input	**DO NOT DO THIS**
Screen messages	REM DISPLAY.MESSAGE
Screen header	UPDATE.HEADER
Screen printing	**DO NOT DO THIS**
Fatal errors	FATAL.ERROR
Report writing	PRINTER.ON PRINTER.OFF PRINTER.CLOSE
Displaying enquiries	ENQ.DISPLAY
Selects	Use the appropriate DAS (See Data Access Services) or use EB.QUERY.BUILDER
Amount formatting	EB.ROUND.AMOUNT
Redisplay fields	REFRESH.FIELD
Error messages	ERR
Field Display	REFRESH.FIELD

Data Access Services

Query names for a DAS should start with "das" followed by the name of the application and, finally, the name of the query. Query names should be in lower case with the first character of the second word onward in upper case. (dasApplicationQuery)

For example a query on application *SEC.TRADE* which selects all deals traded today would be structured dasSecTradeAllDealsToday.

This format is currently only for DAS queries. Please see the T24 Application Development User Guide for further details on generating and using DAS queries.

Data Access Services (DAS) consolidated all queries (i.e. SELECT operations) that are performed against a table/application. This centralises all query processing for a given table in one place and removes any query language from the code. Each query is expressed in a simple meta-query language (as per ENQUIRY) that defines the fields, operands, data and joins between criteria (AND / OR).

When executed, the meta-query language generates the specific query language depending on the target database (JQL, SQL, xQuery, etc.)

This approach allows control on the DAS routines and prevents poor queries being defined. Each query in the DAS is named and has a description, and the DAS itself allows discovery of the queries that are supported. There is also a facility to cache the results of static data.

The name of the DAS is DAS.MYAPPLICATION, where MYAPPLICATION is the name of the application, e.g. DAS.EB.LOOKUP

Below are several examples of DAS being invoked:

```
...
$INSERT I_DAS.EB.LOOKUP
...
THE.LIST = dasEbLookupItems
THE.ARGS = VIRTUAL.TABLE
CALL DAS('EB.LOOKUP', THE.LIST, THE.ARGS, '')

...
$INSERT I_DAS.COMMON
$INSERT I_DAS.ACCOUNT
...
THE.LIST = dasAccountGroupCcyById
THE.ARGS = CURRENCY
dasMode = dasReturnQuery
CALL DAS('ACCOUNT', THE.LIST, THE.ARGS, '')
LIST.PARAMETER = ''
LIST.PARAMETER<3> = THE.LIST
CALL BATCH.BUILD.LIST(LIST.PARAMETER, '')

...
$INSERT I_DAS.COMMON
$INSERT I_DAS.COMPANY
...
THE.LIST = dasAllIds
THE.ARGS = ''
dasMode = dasReturnDescription
CALL DAS('COMPANY', THE.LIST, THE.ARGS, '')
CRT THE.LIST<1> ; * The query
CRT THE.LIST<2> ; * The description
```

Please see the T24 Application Development User Guide for further details.

GUI/Desktop/Browser/STP/OFS Compatibility

Due to the highly structured nature of the T24 application interface there is little difference when programming for Browser, Classic or GUI. However the GUI is standard client server architecture that relies on structured messages for its communication, hence it cannot respond to direct ‘screen’ input and output commands. Straight through processing (STP) utilises OFS to generate transactions without user intervention.

OFS and Browser are completely stateless, any ‘conversation’ will stop them working. Direct input through TXTINP or REM should not be used.

You must use the routines DISPLAY.MESSAGE and UPDATE.HEADER otherwise your code will not work with the T24 Graphical User Interface. Commands such as INPUT, PRINT and CRT must never be used in a routine / program.

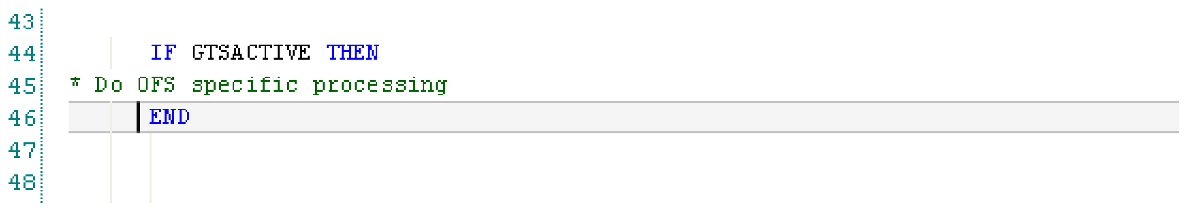
PRINT and CRT statements will simply display nothing in the GUI, whilst the INPUT statement will have severe implications and may cause the session to hang and/or terminate abnormally.

You should never use the INPUT.BUFFER (used as a keyboard type ahead) variable for defaulting data or navigation commands e.g. moving to a new field, expanding a multi-value etc. INPUT.BUFFER cannot be interpreted by the GUI. Use the standard mechanisms of T.SEQU, REFRESH.FIELD and REBUILD.SCREEN to refresh the screen, although the latter should be used sparingly as it means that all field definitions and all data are resent to the GUI.

There are ‘standard’ methods for determining whether an application or subroutine is running in any specified environment:

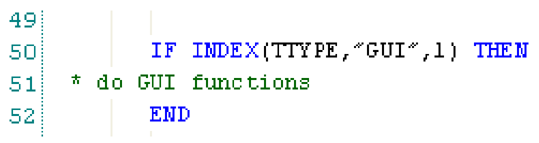
- For OFS the common variable GTSACTIVE will be set.
- For GUI the common variable TTYPE will contain the text GUI.
- For Browser the common variable GTSACTIVE will be set.

Therefore if an application is to execute any code specific to OFS then the relevant call can be made if GTSACTIVE is set as follows:



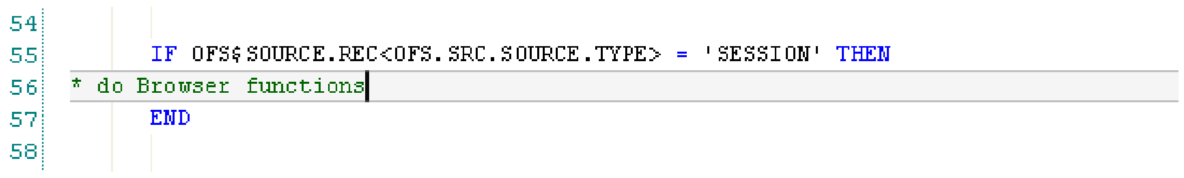
```

    IF GTSACTIVE THEN
* Do OFS specific processing
    END


```

Similarly functionality only applicable to the GUI interface can be executed as follows:



```

         IF INDEX(TTYPE,"GUI",1) THEN
* do GUI functions
         END
```

Browser can be identified as follows:



```

         IF OFS$SOURCE.REC<OFS.SRC.SOURCE.TYPE> = 'SESSION' THEN
* do Browser functions
         END

```

jBASE compatibility issues

In order to keep T24 compatible with the jBASE platform there are certain restrictions that will apply to code development within T24. These have been listed below:

Keywords used as Variables

jBase does not allow the use of keywords as variables e.g. FUNCTION used as a variable. The jBASE compiler does not allow this – see the list of keywords below that must not be used as variables.

ABORT	ABS	ACOS	ADDS
ALPHA	AND	ANDS	APPEND
APPENDING	ARG	ASCII	ASIN
ASSIGNED	AT	ATAN	ATKEY
ATVAR	BEFORE	BEGIN	BFUNC
BITAND	BITCHANGE	BITCHECK	BITLOAD
BITNOT	BITOR	BITRESET	BITSET
BITXOR	BLOB	BREAK	BY
CALL	CALLC	CALLJ	CAPTURING
CASE	CATS	CE	CFUNC
CHAIN	CHANGE	CHAR	CHARS
CHDIR	CHECKSUM	CLEAR	CLEARCOMMON
CLEARDATA	CLEARFILE	CLEARSELECT	CLOSE
CLOSESEQ	COL1	COL2	COLLECTDATA
COMMON	COMPARE	CONTINUE	CONVERT
COS	COUNT	CRC	CRT
CTYPE	DATA	DATE	DCOUNT
DE	DEBUG	DECLARE	DECRYPT
DEFB	DEFC	DEFCPP	DEFFUN
DEL	DELETE	DELETELIST	DELETESEQ
DICT	DIM	DIR	DISPLAY
DIVS	DO	DOWNCASE	DQUOTE
DTX	EBCDIC	ECHO	EDYN
ELSE	ENCRYPT	END	ENTER
ENVFUNC	EOL	EQ	ERROR
ERRTEXT	EXEC	EXECUTE	EXISTING
EXIT	EXP	EXTRACT	FADD

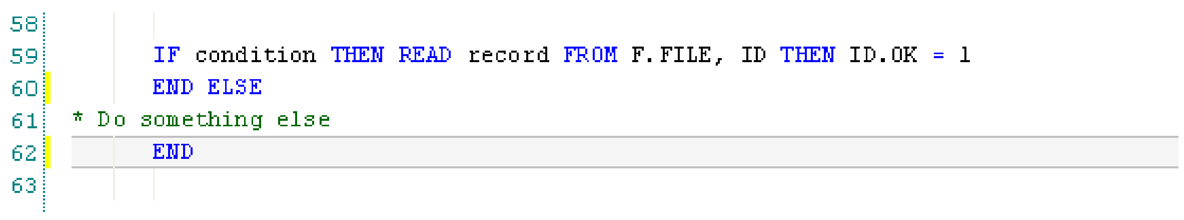
FCMP	FDIV	FFIX	FFLT
FIELD	FIELDS	FILELOCK	FIND
FINDSTR	FLOAT	FMT	FMUL
FOLD	FOOTING	FOR	FORMLIST
FROM	FSUB	FUNCTION	GE
GES	GET	GETCWD	GETENV
GETLIST	GETX	GO	GOSUB
GOTO	GROUP	GROUPSTORE	GT
HEADING	ICONV	IF	IN
INDEX	INDICES	INMAT	INPUT
INPUTCLEAR	INPUTERR	INPUTNULL	INPUTTRAP
INS	INSERT	INT	IOCTL
ITYPE	KEY	LE	LEN
LN	LOCATE	LOCK	LOCKED
LOC_SDYN	LOOP	LOWCASE	LOWER
LP	LT	MAT	MATBUILD
MATCHES	MATCHFIELD	MATPARSE	MATREAD
MATREADU	MATVAR	MATWRITE	MATWRITEU
MAXIMUM	ME	MINIMUM	MM
MOD	MSG	MSLEEP	MULS
NE	NEG	NEGS	NEXT
NOT	NULL	NUM	OBJEXCALLBACK
OCONV	OFF	ON	ONGOSUB
ONGOTO	OPEN	OPENDEV	OPENINDEX
OPENPATH	OPENSEQ	OPENSER	OR
ORS	OUT	PAGE	PASSDATA
PASSLIST	PAUSE	PE	PERFORM
PERROR	POSCUR	PP	PRECISION
PRINT	PRINTER	PRINTERR	PROCREAD
PROCWRITE	PROGRAM	PROMPT	PUTENV
PWR	QUOTE	RAISE	READ
READLIST	READNEXT	READONLY	READPREV
READSEQ	READT	READTX	READU
READV	READVU	READX	READXU
RECORDLOCKE	RECORDLOCKU	REGEXP	RELEASE
REM	REMOVE	REPEAT	REPLACE
RETURN	REUSE	REWIND	RND

RTNDATA	RTNLIST	SADD	SCMP
SDIV	SDYN	SECTION	SELECT
SELECTE	SELECTINDEX	SELECTINFO	SEND
SENDX	SENTENCE	SEQ	SETTING
SIN	SLEEP	SMUL	SORT
SOUNDEX	SPACE	SPLICE	SPOOLER
SQL	SQRT	SQUOTE	SSUB
STATIC	STATUS	STEP	STOP
STR	STRING	SUBR	SUBROUTINE
SUBS	SUBSTRINGS	SUM	SUMMATION
SWAP	SYNC	SYSTEM	TA
TABLE	TAN	TE	THEN
TIME	TIMEDATE	TO	TRANS
TRANSABORT	TRANSEND	TRANSQUERY	TRANSTART
TRIM	TRIMB	TRIMF	UNASSIGNED
UNFILTERED	UNLOCK	UNTIL	UPCASE
USING	VAR	VARCHAR	WAITING
WAKE	WEOF	WEOFSEQ	WHILE
WITH	WRITE	WRITEBLK	WRITELIST
WRITEP	WRITEPU	WRITESEQ	WRITESEQF
WRITET	WRITETX	WRITEU	WRITEV
WRITEVU	WRITEX	WRITEXU	XTD

See http://www.jbase.com/ for an up to date list.

Ambiguous THEN & ELSE Statements

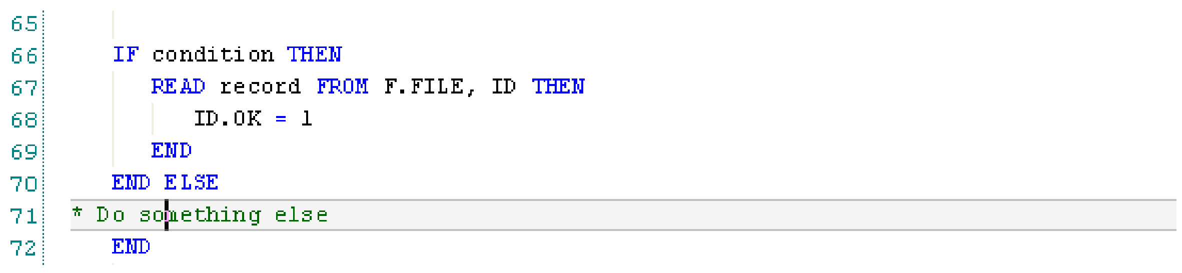
The constructions of THEN and ELSE clauses can be very complicated and the two compilers can treat ambiguous statements differently, for example:



```

          IF condition THEN READ record FROM F.FILE, ID THEN ID.OK = 1
          END ELSE
* Do something else
          END

```

Should the ELSE go with the READ or the IF?

Basically this is poor coding technique and instances like this have been re-written so that they are both readable and non-ambiguous, hence this should become:



```
   IF condition THEN
      READ record FROM F.FILE, ID THEN
         ID.OK = 1
      END
   END ELSE
* Do something else
   END
```

CHAR, CHARS, SEQ & SEQS

The use of these function calls is supported in jBASE but they cause an incompatibility issue when running in an EBCDIC environment (i.e. the IBM s390). In order to achieve jBASE/s390 compatibility these function calls will be replaced by T24 functions viz. CHARX(), CHARSX(), SEQX() and SEQSX().

C$CHARX() is a dimensioned array, populated in OVERLAY.EX, which will ALWAYS return the ASCII character of the number requested. If running in an EBCDIC environment the EBCDIC characters will be re-ordered in C$CHARX(), hence CHARX(10) will always be a linefeed.

CHARX() is a function, defined in I_COMMON, that will return the character at the position requested from the C$CHARX() array. Hence CHARX() will always use the ASCII table for its interpretation even when running in an EBCDIC environment.

CHARSX() is similarly a function defined in I_COMMON that works off a list as opposed to a single value.

SEQX() is a function, defined in I_COMMON that will return the index position in C$CHARX() of the character supplied. Hence SEQX() will always give you the ASCII code for a character, even when running in an EBCDIC environment.

SEQSX() is similarly a function defined in I_COMMON that works off a list as opposed to a single value.

SUBROUTINE <name>

When coding a subroutine it must be ensured that the <name> portion code is identical to the source code name. This is required to ensure that in jBASE the subroutine gets built with the name with which it is finally being called in the T24 programs.

jBASE compiles programs to executables and libraries. The executables and libraries are stored in lib and bin respectively with the name as used in the routine SUBROUTINE <name> or PROGRAM <name>.

For example, if the program is called as E.GET.ACC.DET, but in the program it is declared as

SUBROUTINE E.GET.ACC.DET.TEST.

The routine will be catalogued as E.GET.ACC.DET.TEST and not as E.GET.ACC.DET. Hence the routine will be not be called correctly and you will encounter a run-time error. Using EB.COMPILE will prevent compilation of subroutines where the subroutine name does not match the record name.

Choice between ‘PROGRAM’ & ‘SUBROUTINE’

Most programs in T24 must be coded to be ‘SUBROUTINE’. In case it is required to call a program from the jBASE prompt or call the program using an EXECUTE statement, then program must be coded as a ‘PROGRAM’. In case the program has been coded as ‘PROGRAM’ the statement used to exit the program must be coded as ‘STOP’ and not ‘RETURN’ since ‘RETURN’ is associated with a ‘SUBROUTINE’ in jBASE.

WRITE ON ERROR or DELETE ON ERROR

In case a programmer needs to bypass T24 subroutines to perform delete (F.DELETE) & write (F.WRITE), the syntax in these statements must avoid the ELSE & THEN verbs and instead make use of only the ON ERROR verb.

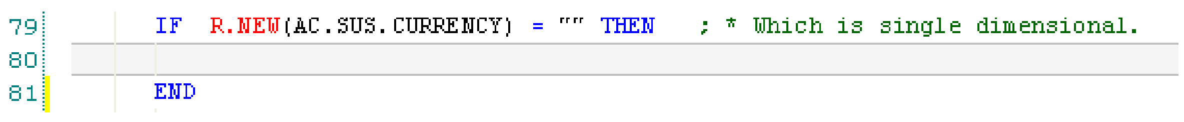
Dynamic Array Usage

Dynamic arrays must always be used with a single ‘<’ & a single ‘>’ character. Any other use of dynamic arrays is illegal in jBASE.

Please be aware that when using dynamic arrays that they become more and more inefficient to store data when the number of values/multi-values becomes large. Even at around 100 the processing will noticeably degrade.

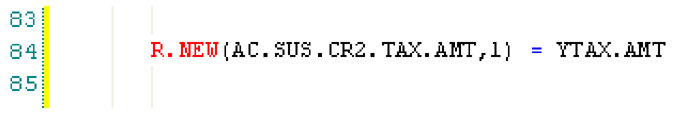
Dimensioned array usage

Dimensioned arrays are defined as single or multi-dimensioned array. The array should be used in same way as it is dimensioned. For example, do not use a multi-dimensioned array as a single dimensioned or vice-versa.



```
        IF  R.NEW(AC.SUS.CURRENCY) = "" THEN   ; * Which is single dimensional.

        END
```

The same arrays as Multi-dimensioned:

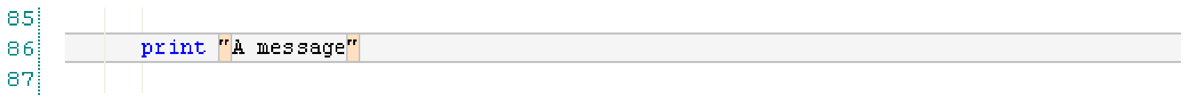


```
R.NEW(AC.SUS.CR2.TAX.AMT,1) = YTAX.AMT
```

Keywords Usage

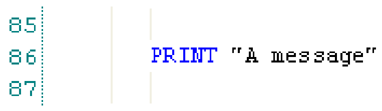
All keywords used must always be coded in capital letters.

For example:



```
print "A message"
```

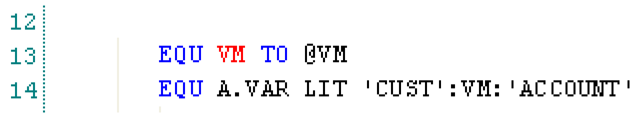
Is invalid and must be coded as



```
PRINT "A message"
```

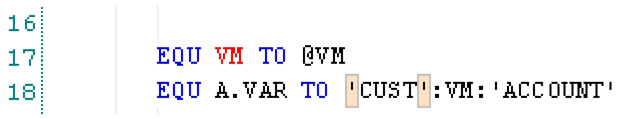
EQU LIT Usage

The use of EQU LIT can become ambiguous.



```
EQU VM TO @VM
EQU A.VAR LIT 'CUST':VM:'ACCOUNT'
```

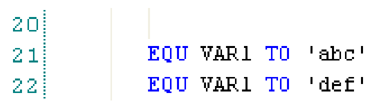
The second line of code is ambiguous and the pre-compiler in jBASE will not convert the VM to @VM thereby giving unexpected results. The code should be re-written as:



```
EQU VM TO @VM
EQU A.VAR TO 'CUST':VM:'ACCOUNT'
```

EQU Usage

jBASE will allow the EQU statement to be used more than once for the same variable. For example:

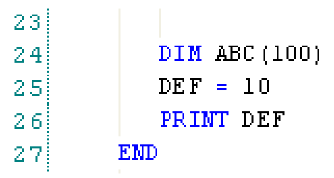


```
    EQU VAR1 TO 'abc'
    EQU VAR1 TO 'def'
```

Care must therefore be taken to ensure that such mistakes are not made, as the jBASE compiler will not report these errors.

Warnings in jBASE

jBASE gives a warning when the dimensioned variable is defined but not used. For example:



```
    DIM ABC(100)
    DEF = 10
    PRINT DEF
END
```

In this case the variable ABC is defined but not used. Hence it gives a warning. This does not cause any harm and hence can be ignored. However, it is recommended that the unwanted code be removed.

It should be noted that the setting of the environment variable JBASE_WARNLEVEL in jBASE can suppress the appearance of run time error messages. See http://www.jbase.com/ for more details.

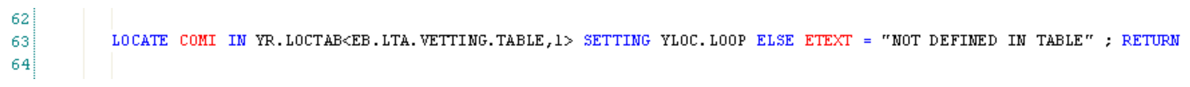
UniVerse Special routines/functions

Certain UV routines beginning with '!' (e.g. !MINIMUM, !MAXIMUM, etc.) are not supported in jBASE.

Since !HUSHIT(), !SLEEP$() are commonly used, these two routines are supported in jBASE, but it is recommended that these programs are not used and replaced by HUSH ON/OFF and SLEEP.

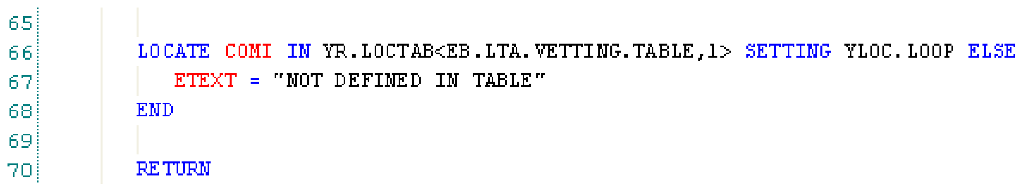
LOCATE statement

The following single line syntax of LOCATE will not work correctly.



```
LOCATE COMI IN YR.LOCTAB<EB.LTA.VETTING.TABLE,1> SETTING YLOC.LOOP ELSE ETEXT = "NOT DEFINED IN TABLE" ; RETURN
```

The correct Syntax is

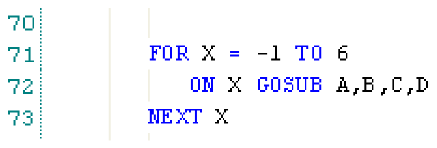


```
LOCATE COMI IN YR.LOCTAB<EB.LTA.VETTING.TABLE,1> SETTING YLOC.LOOP ELSE
   ETEXT = "NOT DEFINED IN TABLE"
END

RETURN
```

Locate () will not work the same way in jBASE. Hence do not use LOCATE(). Always use the LOCATE statement.

ON GOSUB/ON GOTO

In UniVerse, if you use the "ON x GOSUB" command with x outside of the range, you do not get an error; in jBASE you do:



```
FOR X = -1 TO 6
   ON X GOSUB A,B,C,D
NEXT X
```

In UniVerse:

If X < 1 then GOSUB A is invoked

If X > 4 then GOSUB D is invoked

In jBASE:

If X < 1 then the following error message is produced:

```
Branch value of computed GOTO/GOSUB out of range
Trap from an error message, error message name = COMPUTED_LESS
```

If X > 4 then the following error message is produced:

```
Branch value of computed GOTO/GOSUB out of range
Trap from an error message, error message name = COMPUTED_MORE
```

The code should be amended to use a CASE statement; this will aid readability as well as functionality.

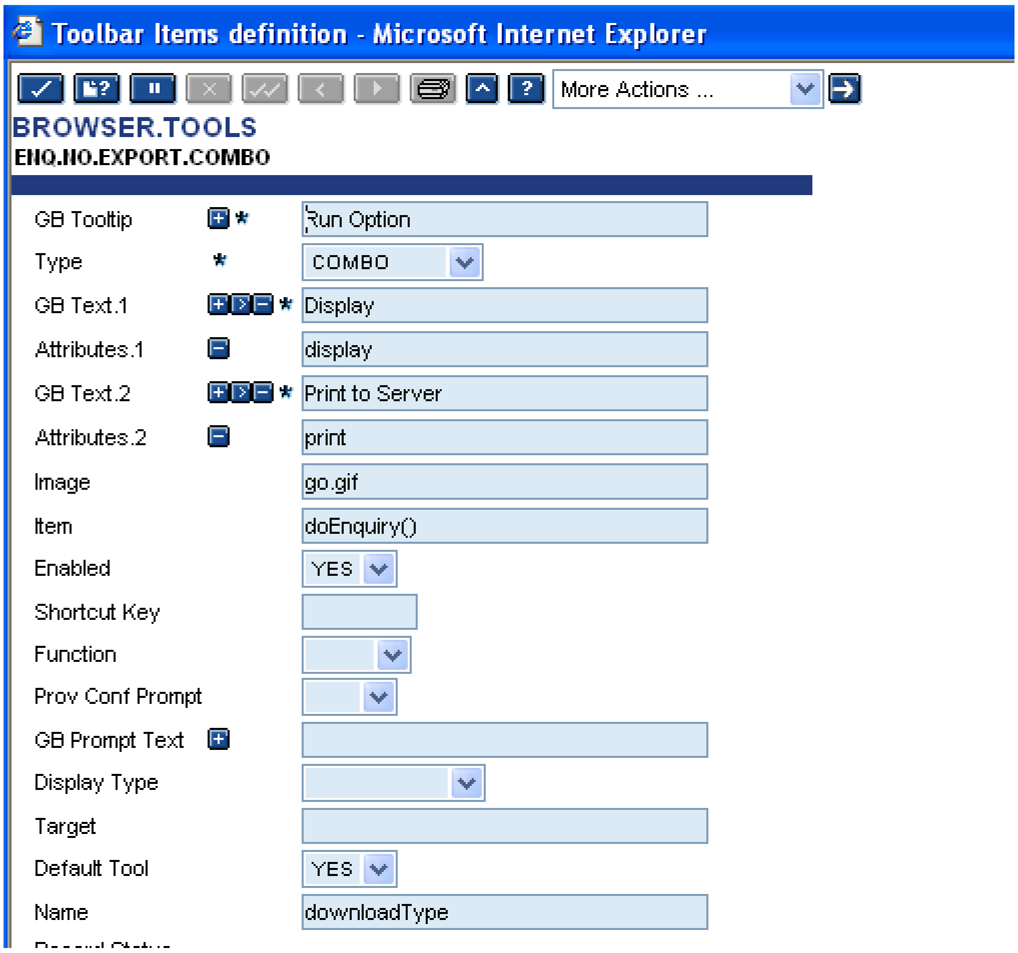
Toolbars in Browser

The creation of Toolbars in Browser is done using two tables. *BROWSER.TOOLS*, which defines the buttons that you wish to use, and *BROWSER.TOOLBAR*, which defines a toolbar as a grouping of individual buttons. There are a number of subroutines to be called to build the toolbars.

Defining the toolbars

BROWSER.TOOLS

This table defines the buttons to be used in any toolbar. Each button is described by a number of fields in the table:



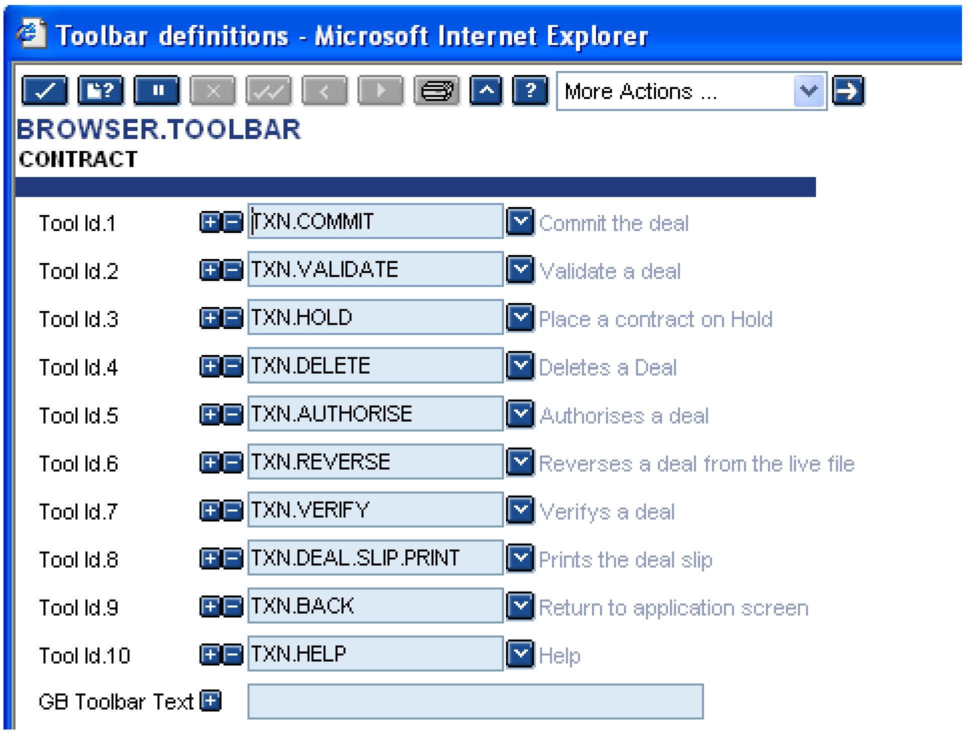
BROWSER.TOOLS
ENQ.NO.EXPORT.COMBO

GB Tooltip	Run Option
Type	COMBO
GB Text.1	Display
Attributes.1	display
GB Text.2	Print to Server
Attributes.2	print
Image	go.gif
Item	doEnquiry()
Enabled	YES
Shortcut Key	
Function	
Prov Conf Prompt	
GB Prompt Text	
Display Type	
Target	
Default Tool	YES
Name	downloadType

See the Browser User Customisation User Guide and helptext for full details of this application.

BROWSER.TOOLBAR

This table defines the toolbar. Each toolbar is made up of a multi value of tools:


Toolbar definitions - Microsoft Internet Explorer

More Actions ...

BROWSER.TOOLBAR
CONTRACT

Tool Id.1	TXN.COMMIT	Commit the deal
Tool Id.2	TXN.VALIDATE	Validate a deal
Tool Id.3	TXN.HOLD	Place a contract on Hold
Tool Id.4	TXN.DELETE	Deletes a Deal
Tool Id.5	TXN.AUTHORISE	Authorises a deal
Tool Id.6	TXN.REVERSE	Reverses a deal from the live file
Tool Id.7	TXN.VERIFY	Verifys a deal
Tool Id.8	TXN.DEAL.SLIP.PRINT	Prints the deal slip
Tool Id.9	TXN.BACK	Return to application screen
Tool Id.10	TXN.HELP	Help
GB Toolbar Text		

See the Browser Navigation User Guide and helptext for full details of this application.

Building the toolbars in your application

To build the toolbars in your application, use the following API:

EB.ADD.TOOLBAR (TOOLBAR.ID, TOOLBAR.LOCATION)

This routine is passed the key to a record on the *BROWSER.TOOLBAR* table. This loads the toolbar into memory. If the toolbar is already in memory then it does nothing. If the Toolbar does not exist on the *BROWSER.TOOLBAR* file then it creates a blank toolbar. If a value is sent in the second field value of *TOOLBAR.ID* then it is used to set the Text for the toolbar or override the text from the *BROWSER.TOOLBAR* file. The second parameter is the location at which the tool will be displayed.

EB.ADD.TOOL (TOOLBAR.ID, TOOL.ID)

This routine is passed the key of a toolbar already in memory and the key of a record on the *BROWSER.TOOLS* table to be added to the toolbar in memory. If the tool is already on the toolbar in memory then it does nothing. Adding a tool to a toolbar in memory will not amend the record on the *BROWSER.TOOLBAR* file. Instead of passing the key to a tool a *BROWSER.TOOLBAR* record can be manually built and passed in place of the *TOOLBAR.ID*.

EB.CHANGE.TOOL (TOOLBAR, TOOL, FIELD, NEW.VALUE)

This routine is passed the key of a toolbar already in memory and the key of a tool that is currently on that toolbar in memory. It is also passed a field number and the new value to set that field. For the field number, equates in the *BROWSER.TOOLS* insert should be used (e.g. `BRTL.TEXT` or `BRTL.ENABLED`). If the toolbar is not in memory or the Tool is not on the toolbar that is in the memory then this routine will do nothing.

Handling Context Workflow

When trying to run a new task in T24 or giving the user an option of new tasks to run based on the current context, there are two functions that should be called:

EB.SET.NEXT.TASK(NEXT.TASK)

This sets the value of the next task to be run. The NEXT.TASK should be passed in like a T24 command line, e.g. “ACCOUNT I DBL”. This should only be set after a commit.

EB.CREATE.CONTEXT.WORKFLOW(DISPLAY.TEXT, INPUT.OPTIONS, OPTION.COMMANDS)

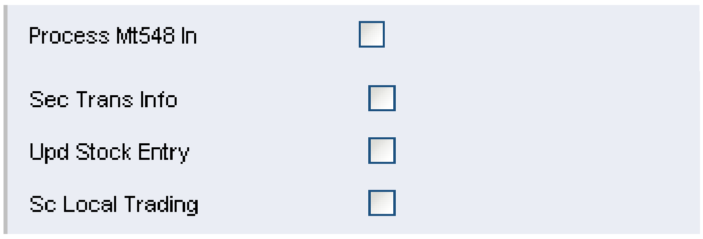
This subroutine is passed the options to be given to the user upon committing a deal and will then run an action depending on the choice. The input can be in two ways. The first is to set the Display for the Option in DISPLAY.TEXT. Then the captions for the different options should be set in a dynamic array delimited by field markers in the INPUT.OPTIONS parameter. The corresponding actions to run should be in a correlating dynamic array in OPTION.COMMANDS. The format of these commands should be from the T24 command line, e.g.: “ACCOUNT I DBL”. Alternatively if you are running in Browser the key to a toolbar can just be passed in the first parameter. The actions on the tools on the toolbar should be DO.DEAL.

Radio Buttons/Check Box/Combo Box

When the T array is set for choosing an option from a limited list using T<2> the following conventions are followed.

If there are one or two options and the options are a combination of the following strings “Y”, “YES”, “N” or “NO” then this will be rendered as a check box. For example;

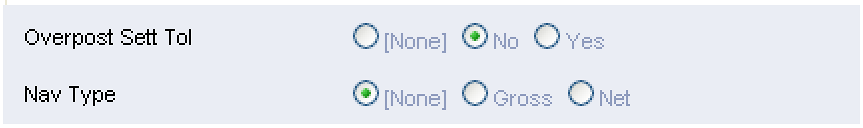
```
Z+=1 ; F(Z) = "PROCESS.MT548.IN" ; N(Z) = "3" ; T(Z)<2> = "YES_"
Z+=1 ; F(Z) = "SEC.TRANS.INFO" ; N(Z) = "3" ; T(Z)<2> = "NO_"
Z+=1 ; F(Z) = "UPD.STOCK.ENTRY" ; N(Z) = "3..C" ; T(Z)<2> = "YES_"
Z+=1 ; F(Z) = "SC.LOCAL.TRADING" ; N(Z) = "1" ; T(Z)<2> = "Y_"
```



Process Mt548 In	☐
Sec Trans Info	☐
Upd Stock Entry	☐
Sc Local Trading	☐

If the above criteria are not met, i.e. two or more values, optional input or a blank option, and the field is not marked as a combo box in the version then radio buttons will be displayed. For example;

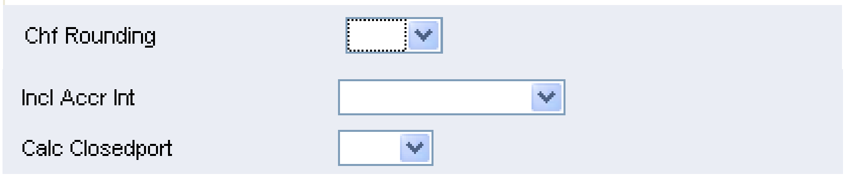
```
Z+=1 ; F(Z) = "OVERPOST.SETT.TOL" ; N(Z) = "3..C" ; T(Z)<2> = "YES_NO"
Z+=1 ; F(Z) = "NAV.TYPE" ; N(Z) = "5..C" ; T(Z)<2> = "NET_GROSS"
```



Overpost Sett Tol	[None] [✓ No] Yes
Nav Type	[✓ [None]] Gross Net

For all other options a combo box will be displayed. For example;

```
Z+=1 ; F(Z) = "CHF.ROUNDING"; N(Z) = '3' ; T(Z)<2> = "YES_NO_"
Z+=1 ; F(Z) = "INCL.ACCR.INT" ; N(Z) = "13" ; T(Z)<2> = "NEXT.WORK.DAY"
Z+=1 ; F(Z) = "CALC.CLOSEDPORT" ; N(Z) = "3" ; T(Z)<2> = "YES_NO_"
```



Chf Rounding

Incl Accr Int

Calc Closedport

Close of Business Routines

BATCH Records

All new *BATCH* records must also have an equivalent record in the *BATCH.NEW.COMPANY* application.

Code Requirements

- All new batch processes should be written to use multi-threading.
- Close of Business routines must be structured according to the Close of Business routines section in the Development guide.
- The same programming standards as for online routines apply and additionally:

If a routine needs to know that it has been called during the COB procedure the flag RUNNING.UNDER.BATCH must be tested. Do not test the *SPF* status, this will be set as soon as the Close of Business has been initiated, hence the routine will not know if it has been invoked from the 'action line' or run as part of the batch.

- Do not call template programs from the batch. Isolate the functionality you require into a subroutine and call the subroutine from the Close of Business and the template program.
- Never prompt during the Close of Business. Use the OCOMO subroutine to record progress of the job (take care when large volumes can be encountered).
- Do not use HUSH or !HUSHIT, the output from 'commands' is very useful when analysing an COB failure.
- Always use FATAL.ERROR to terminate your batch job when an error condition occurs. Never use STOP.
- The FATAL.ERROR shall be called with the proper TEXT set such that the error logged in EB.EOD.ERROR is helpful for further analysis.
- End of day reports shall be attached to the EB.PRINT either as routines or repgens or enquiries.

Summary of Standard Multi - Thread process

It is **MANDATORY** to write multi threaded batch jobs so that there is minimal impact on the performance and we optimally use the processing capabilities of the hardware. Multi-threading of routines is mandatory to ensure that transaction management is provided by the system.

Main Process

Name: **xxxx(list_record_id)**

xxxx must be the name defined in *PGM.FILE* and the name used in the *BATCH* record, the field `BATCH.JOB` in the *PGM.FILE* for a standard multi-thread process should be

@BATCH.JOB.CONTROL. If the field is left null, then the system defaults it to @BATCH.JOB.CONTROL.

list_record_id is the key extracted from the list record which is keyed on sequence.

This routine should be processing the single piece of transaction of the job thus ensuring better split between different agents. The routine is invoked by the system with the transaction management wrapped around it. Thus the routine should not be performing a heavy load and should avoid Selects/Clearfiles.

Load Process

Name: **xxxx.LOAD**

This mandatory process is required to set up the common area used by the main process. Typically a local common will be created for each main process. This routine is invoked once per agent and should avoid any locks and any writes to database.

Select Process

Name: **xxxx.SELECT**

Mandatory process to build the LIST file that the main processes will handle. Called after the LOAD routine. The list file must be built by calling The BATCH.BUILD.LIST (ListParameters, IdList):

ListParameters<1>	Must be null.
ListParameters<2>	Extract File Name a file used to hold ids. The work file list will be built by a simple copy from this file. Should not include the company mnemonic.
ListParameters<3>	Extract Selection – a list of selection criteria to be supplied to select and build from the extract file. The command can supply SELECT or SSELECT if the ordering is important.
ListParameters<4>	Last key to the list file, used when multiple calls are made to BATCH.BUILD.LIST.
ListParameters<5>	Total number of keys processed, used when multiple calls are made to BATCH.BUILD.LIST
ListParameters<4>	The number of record ids written to a work list record. The system Default is 1, but where large volumes are expected this should be increased to a larger number to reduce the overhead of building the work file. Be aware of the performance implications of increasing this size, since all ids in the list will be processed as a single transaction and will hold record locks for this duration. System will limit this to 200 if programmers set this to greater than 200.
ListParameters<7>	Flag to determine call to .FILTER routine, set to 1 if call required.
IdList	A list of ids prepared by the SELECT program used to build the list file

The list file is allotted dynamically based on the pool available. The select routine is invoked per job. But in case of jobs where the requirement is to perform different operations in sequence with every operation having an ability to run in parallel, the CONTROL.LIST The CONTROL.LIST common variable can be populated the select routine for the first time and the system invokes the SELECT routine for the n number of times and thus phasing out different stages of the job within which every stage can have parallel processing.

Filter Process

Name: **xxxx.FILTER(list_record_id)**

This optional routine is invoked by BATCH.BUILD.LIST when LIST.PARAMETERS<7> is set. It is used to filter the Id's selected before writing to the list file and should not under any circumstances perform any I/O operations. Any decisions should be based solely on the Id and variables loaded into the named common area. If the Id should not be included in any processing then set the subroutine argument list_record_id to null.

Post Process

Name: **xxxx.POST**

Optional process to handle any processing required after the list file and all multi-threading has finished. This process must be defined in *BATCH*.

Template Programs

- Every file created must have an associated template program.
- Use the correct template program for the file type.
- Never modify the template control code.
- The basic minimum for a new application will be the application and an associated FIELD.DEFINITIONS.

See the T24 Application Development User Guide for further information.

Performance Considerations

- Use the faster operators ":=" "+=" etc.
  e.g.

```
TOTAL.VAL += SUB.TOTAL
```

instead of

```
TOTAL.VAL = TOTAL.VAL + SUB.TOTAL.
```

- Avoid large dynamic arrays (greater than 1000 bytes).
- Always use ":=" instead of <-1> when building a large dynamic array. There is little difference when performed at FIELD level, but this is expensive on multi and sub values.
- Construct case statements so that the most often used condition is at the top.
- Do not perform counts in the controlling section of a loop:

  e.g.

  ```
  FOR FOR CNT = 1 TO DCOUNT(ARRAY, FM)
  ```

  should be two statements:

  ```
  NO.ITEMS = DCOUNT(ARRAY,FM)
  FOR CNT = 1 TO NO.ITEMS
  ```

  The system will evaluate the DCOUNT for every pass of the FOR NEXT loop, which can be very expensive with large arrays, where it only needs to be evaluated once.
- Do NOT use DBR to read single fields from a record as this reads the whole record, it is better to read the entire record once and extract the values required. A typical subroutine will access more than value from a record, and even if it does not then further enhancement will be easier if the entire record is available.
- Ensure that new files created have a sensible modulo. Do NOT release new file control records with a modulo of 3 or 9731, try to work out a projected record size and base the size on a database with a reasonable volume.
- Make sure that operations are not needlessly repeated within a loop, for example avoid re-reading the same parameter record every time a loop is executed.
- Do NOT select files that are known to be likely to grow large in installations, use concat files or other alternative methods of finding the records required. When developing new applications / processes. If a process will be dependent upon selection of a file that will grow large, build an alternative access method, e.g. a concatfile or use indexes.

Example files include:

- STMT.ENTRY
- CATEG.ENTRY
- RE.CONSOL.SPEC.ENTRY
- LIMIT.TXNS
- SC.POS.ASSET

- Be very careful in the use of concat files. A poorly designed concat file can result in a performance overhead when updated due to the large record size created and system locks if the key is a poor choice. E.g. the concat file LMM.CURRENCY is a poor file since the key is likely to be common with the majority of contracts, this has the effect of allowing only a single contract to be entered at any one time for a given currency, and is likely to contain massive records (e.g. for local currency). If a concat file is likely to contain more than 1000 ids in it a different structure / process should be used.
- Do not re-invent the wheel. Use standard system routines and functions where available, these will have been optimised for performance.
- Avoid unnecessary select statements; if a record key can be obtained using direct access, a select is an unnecessary overhead. There have been examples where a file is selected with a specific key, put into a select list, and then used to read the record.
- Use SUM etc to add all multi-values in a field together. Do not use a loop to process each multi-value separately.
- Do not use SAVING UNIQUE on selection criteria this performs badly on large systems. Also note there is no mechanism for this in the DAS system.
- Allow for non-numeric sequence numbers in transaction ids. This can occur when the AUTO.ID.START application is setup to use unique ids. This allows applications to generate new ids without reference to (and locking of) a single record in F.LOCKING which causes a potential bottleneck. It also allows for more that 99,999 transactions per day which is important in volume applications.
- Conversely do not code for numeric sequence numbers in transaction ids as this will cause problems is the above method is chosen to allocate transaction ids.

EB.COMPILE – On-Site development

EB.COMPILE has been designed for jBASE users to automatically set JBCDEV_BIN and JBCDEV_LIB when cataloguing programs. The following rules are applied:

If JBCDEV_BIN/LIB is currently set to globusbin/lib (the default):

If you are compiling from BP or GLOBUS.BP, globusbin/lib will be updated.

If you are compiling from any other BP for example MYBANK.BP, bin/lib will be updated.

If JBCDEV_BIN/LIB is set to anything other than globusbin/lib:

If you are compiling from BP or GLOBUS.BP (wherever you are), globusbin/lib will be updated.

If you are compiling from any other BP for example MYBANK.BP, the bin/lib specified in JBCDEV_BIN/LIB will be used.

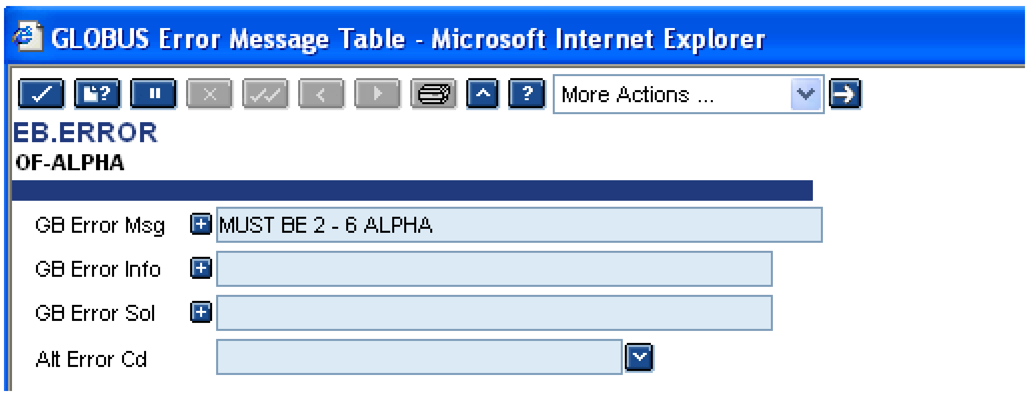
Therefore, GLOBUS.BP and BP should only be used for T24 core programs, by Temenos. BP should not be used for any on-site developed software, as globusbin and globuslib are overwritten by the T24 upgrade process. You should therefore adopt your own naming convention for your program directories, e.g. TEST.BP, or ABC.BANK.BP.

EB.COMPILE should ALWAYS be used to compile T24 programs, it has been adapted to use the $INSERT records from GLOBUS.BP automatically without the need to copy to the local BP file and perform some rudimentary checks on the code.

ERROR MESSAGE HANDLING

EB.ERROR application

This table defines the error messages that are produced. All error messages used should be defined in this table. Hard coded text should never be used.



EB.ERROR
OF-ALPHA

GB Error Msg	MUST BE 2 - 6 ALPHA
GB Error Info	
GB Error Sol	
Alt Error Cd	

Figure 1 – EB Error Message Table

See the System Tables User Guide for a full description of this application.

Triggering an Error Message

The subroutine called **ERR** is invoked to present the Message to the user. To present the message the following process needs to be followed:

1. Set the elements of **COMMON** Variable **E** to:

E<*N*>	DESCRIPTION
1	Error Code

	Will be replaced by the Error Message during the course of the processing e.g. OF-ALPHA.
2	Data to replace the & (if used) in element 1. Data is Value Marker – VM (char 253) separated.
3	Name of program / subroutine
4	Developers comment – free text field
5	XX.LL.ERROR.INFO - Error message definition
6	XX.LL.ERROR.SOL – Error message solution
7	Application name that triggered the error
8	ID to the record in F.EB.ERROR table

Key to above table:

Elements of the Dynamic Array to be set-up by Developer.
Optional.

2. Set COMMON Variables, **AF**, **AV**, **AS** (if required) with field, multi-value and sub-value positions

3. Make the CALL to **ERR**. ERR will establish if the user is working in Globus Browser or Classic.

Minimum Testing Requirements

There are no rules that can cover all eventualities for testing, however there are a minimum number of tests that should, but not always be carried out. For example if any new fields are added to an application or the check fields or cross validation is changed then a simple test is to enter a new transaction and immediately hit F5. Additionally if a change has been made to a batch process then a full batch run should be done.

jBASE Testing

When testing in jBASE it is mandatory to set the environment variable JBASE_WARNLEVEL to 4 prior to running any tests. This will ensure that any errors are displayed. Other settings for this variable can suppress error messages. See http://www.jbase.com/ for more details.

Code coverage

If a test has been written to either process or ignore a transaction based on a set of criteria, then a minimum number of transactions will be set by the tests in the code. At least two transactions should be entered to ensure code coverage, one should pass the test, the other should fail the test. Obviously if there are many test criteria, then the number of transactions required to ensure code coverage will multiply.