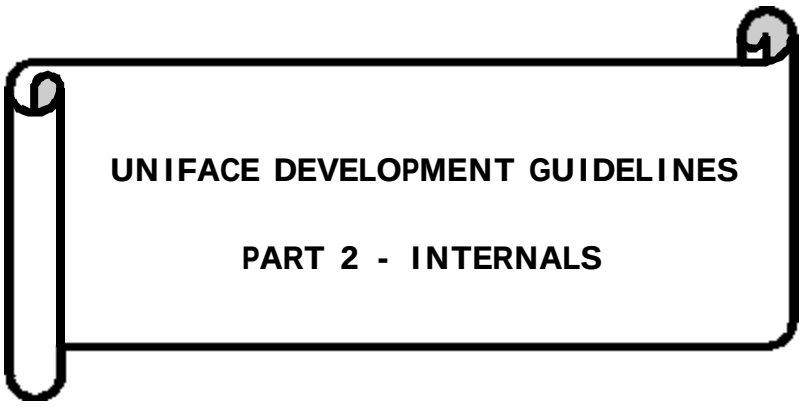


UNIFACE DEVELOPMENT GUIDELINES

PART 2 - INTERNALS

Author:	A J Marston
Date Created:	01 July 1999
Date Changed:	24 February 2002
Version:	04.008.000

NOTICE

TABLE OF CONTENTS

TABLE OF APPENDICES

Amendment History

01.000.000 to 03.004.000 April 1995 to May 1999
Versions produced in a previous life.

04.000.000 July 1999
Total Rewrite.

04.001.000 18th July 1999
Made the following changes to include Column Buttons:-
a) Amemded section *3.3.6 Button Bars*.
b) Added section *3.3.6.3 Column Bar*.
c) Added fCOLUMN_BUTTON to *Appendix A Custom Widgets*.
d) Added COLUMN_BUTTON to *Appendix C Field Interface Templates*.
e) Added COLUMN_BUTTON to *Appendix D Field Syntax Templates*.
f) Added COLUMN_BUTTON to *Appendix E Field Layout Templates*.
g) Added COLUMN_BUTTON to *Appendix F Field Templates*.
h) Added COL_BUTTONS to *Appendix G Global Procedures*.

04.002.000 19th December 1999
a) Changed prefix of valrep lists in message file entries from 'list_' to 'v_'.
b) Updated section on SELECTDB to include note regarding Object Services.
c) Added section *Appendix N: Global Constants - Standard Entries.*
d) Added section *Appendix O: Include Procs - Standard Entries.*
e) Added section *4.7.7 Error Handling with Self-Contained Services*.
f) Added procs ENTITY_LOAD and ENTITY_UNLOAD to *Appendix G.*

04.003.000 27th November 2000
a) Added DEFAULT_LANGUAGE to *Appendix G Global Procedures.*
b) Added READ_INNER_ENT to *Appendix G Global Procedures.*

04.004.000 24th January 2001
a) Added section *4.21Application Startup and Closedown*.
b) Added SAVE_VARIATION to *Appendix G Global Procedures.*
c) Updated *Appendix P:Checklist for creating a new Application*.

04.005.000 22nd April 2001
a) Changed Copyright page.
b) Added CHK_TRAN_ACCESSQ, IGNORE_MESSAGE, PRINT_LIST and SOUNDEX to *Appendix G Global Procedures.*

04.006.000 26th November 2001
a) Changed global proc LMK_TRIGGER to LMK_PROC.
b) Changed global proc VLDK_TRIGGER to VLDK_PROC.
c) Added global procs ENCRYPT and DECRYPT.
d) Added global procs FRGF_PROC and FRLF_PROC.
e) Added include proc LMK_PROC.
f) Added include proc VLDK_PROC.
g) Added include proc VLDF_TRIGGER.
h) Added include proc VLDO_TRIGGER.
i) Changed *Appendix J Global Messages*.
j) Changed section *3.2.6.1 Entity Triggers*.

04.007.000 2nd January 2002

a) Changed *4.15.1 Show Help*.
b) Added CHK_TAB_ACCESS to *Appendix G Global Procedures*.
c) Changed LAUNCH_TAB_PAGE in *Appendix G Global Procedures.*
d) Changed HELP_PROC in *Appendix G Global Procedures.*

04.008.000 24th February 2002

a) Changed *Appendix F: Field Templates* to include TIME_FROM, TIME_TO, VALUE_FROM and VALUE_TO.
b) Changed *Appendix G: Global Procedures* to include AUDIT_BEFOREPROC, AUDIT_AFTERPROC, AUDIT_EXCLUDE and GET_SESSION_DATA.
c) In *Appendix G: Global Procedures* changed OK_PROC, STORE_PROC, STOREQ_PROC, COMMIT_PROC, ROLLBACK_PROC

1. INTRODUCTION

1.1 PURPOSE

This document describes a set of standards and guidelines that could be used for the development of computer systems using the UNIFACE 4th Generation Language deployed on a desktop PC running the Microsoft WINDOWS 9x operating system. These Development Guidelines have been split into the following parts: -

⇒ Part 1 deals with the External view of the software (ie: the "look and feel"). This will be of particular interest to the client as it shows which features are currently available, and indicates how they can be used to interact with the computer system.

⇒ Part 2 deals with the Internal workings of the software (ie: how it is actually constructed). This will be of interest to the development team as it gives examples on how the features described in Part 1 can be implemented, and would therefore be of little interest to the client.

⇒ Part 3 documents the Component Templates used in development.

⇒ Part 4 documents an XAMPLE Application which provides a working demonstration of these guidelines.

These guidelines have been put together using the experience gained while developing various systems using UNIFACE. Their purpose is as follows: -

- To identify the various facilities and options that are available to the system designer that can be used to achieve particular objectives.
- To examine the pros and cons of each method so that the optimum one can be chosen for each particular set of circumstances.
- To encourage the production of software that achieves the desired objectives as efficiently as possible and in a user-friendly manner.
- To encourage the production of software that has a consistent “look and feel” between its individual components so that the user does not have to go through a different learning curve with each component.
- To encourage the production of software that has a consistent “look and feel” with other software products that may be encountered by the user so that he can switch from one product to another and feel that he is in familiar, not alien territory. The Microsoft Windows Graphical User Interface (GUI) contains many “controls” that are used in a standard fashion within every software product developed for that platform. UNIFACE provides “widgets” which match a subset of these controls so that the same functionality can be provided.
- To encourage the construction of software using common techniques and, where possible, common components or routines so that they can be easily maintained and enhanced by any member of the development team, and not just the original author.
- To discourage the use of non-standard components, or features that are undocumented and/or unsupported, so that the software can be easily upgraded to subsequent releases of UNIFACE or other supporting software.

1.2 SOFTWARE VERSIONS

This document has been constructed with the following software versions in mind: -

⇒ UNIFACE version 7.2.06

⇒ Microsoft WINDOWS 95/98/ME/NT

There may be differences with the versions of Uniface that are available on other platforms, but they are outside the scope of this particular document. Certain widgets (controls) are not supported on earlier versions of Microsoft Windows, and screens designed for a GUI will not perform very well if deployed on a Character-based User Interface (CHUI).

New versions of UNIFACE may be released from time to time. Each release may contain new features, changes to existing features, or even the deletion of redundant or superseded features. Before any new release can be successfully deployed the contents of this document may need to be reviewed for accuracy.

1.3 COMPUWARE DEVELOPMENT STANDARDS

Compuware have produced a document titled *UNIFACE Seven Standards* [1](dated 22nd April 1998, part number 1051357200) which contains a great deal of sound advice. It identifies every object that can be encountered within UNIFACE and provides the following information:

- The recommended naming standard for the object type, with the following attributes:

Naming rule	This indicates how the name is determined.
Recommended maximum length	The upper limit for the length of the object name. This can deviate from the length restriction as imposed by UNIFACE.
Unique within	Indicates any uniqueness constraints, eg: is the name of the object unique within the application model, the information system, or perhaps unique within the whole organisation?
Example	These may show several examples of applying the naming conventions, specifically for names with a complex syntax.

- Any restrictions that may apply:

UNIFACE maximum length	This is the upper limit on the object name that is imposed by UNIFACE.
Reserved words from	Identifies potential conflicts with names used by other environments.
Allowed characters	Lists the characters that can be used for the name, as allowed by UNIFACE. Frequently this will be a combination of letters (A-Z), digits (0-9) and underscores (_).
Others	Any other restrictions which may be relevant.

- Remarks

There may be a choice between several alternatives. This section gives the reason for the Rule and supplies some additional considerations and alternatives.

The following sections will not duplicate the information that is contained within the Compuware document, but will identify the extensions that have been adopted by the author of this document.

[1] This has since been superceeded by a document titled *UNIFACE Naming Conventions* (dated July 1999, part number 151010572-00)

2. OBJECT NAMING STANDARDS

2.1 GENERAL

2.1.1 The System (or Application)

1. The maximum length of this code must be less than the maximum length of the object name to which it will be prefixed.

2. Each Application (or System) will have its own name, which can be reduced to its initials to yield a system mnemonic or OWNER_CODE. This should preferably be two characters long, but no more than three. For example, the MENU system would be reduced to MNU.

3. This mnemonic should be used for the following:-
   - The name of the Application Model.
   - The name of the Application Library.
   - The prefix for all Component names.

4. If a large system needs to be broken down into a series of subsystems, the third character within this OWNER_CODE should be altered (or added) so that the first two characters will be consistent within the system as a whole.

5. If you develop different systems for different clients (eg: within a software house) each system should have its own development environment so that all data, initialisation and assignment files can be kept separate from other systems. Each system tends to be referred to as a Project, therefore OWNER_CODE is interchangeable with Project Code.

2.1.2 Assignment File

1. A minimum of two assignment files should be available with different settings - one for the development environment which obtains objects from the development database, and a second for the live or test environments which obtains objects from the DOL and URR files. Name each file accordingly.

2. If you are using a different assignment file for each application, but do not want to put all the company-wide assignments in every application-specific assignment file, you can create an overall (company-wide) assignment file. This file will contain all *global* assignments. The application-specific assignment file will contain the application -specific assignments for that application. Use the "#FILE assignment_file" statement to include the overall assignment file at the appropriate place with the application-specific file. The overall assignment file should be named as mentioned above.

3. User-specific assignment files could be named after the user's logon name, and should be placed in one central asn-file directory.

4. Section headers must be used in order to take advantage of the partition manager. Available section headers are:

[SETTINGS]	for UNIFACE system settings
[DRIVER_SETTINGS]	for driver-specific settings
[FILES]	to specify the location of non-DBMS files
[PATHS]	to direct paths to DBMS, network or GUI drivers
[ENTITIES]	to direct application model entities to paths
[LOGICALS]	to define logical symbols for use in proc code
[SERVICES_EXEC]	to direct service components to machine nodes
[REPORTS_EXEC]	to direct report components to machine nodes
[WIDGETS]	to define widget properties

2.2 APPLICATION MODEL

2.2.1 Model Name

1. The application model should be named after the system/application to which it belongs (see section *2.1.1 The System (or Application)* for more details).

2. An assignment file setting of "*.<application model name>" maps the functionally grouped entities to one DBMS.

3. If a separate application model is generated for a subsystem it would be better to add on (or change) a suffix rather than invent a new name. Application models which are related should contain the same leading characters (prefix) for ease of identification.

2.2.2 Entity

1. Always refer to the DBMS Specific Guide in the UNIFACE documentation and operating system and DBMS/file system supplier information before naming entities. Some restrict identifiers to 30 characters, while others use only the first 21 characters of a table name when creating procedures.

2. Entity names should always be in the singular (eg: CUSTOMER, not CUSTOMERS). The fact that an entity may contain multiple entries is irrelevant - any multiplicity can be derived from the existence of one-to-many relationships.

3. Entity names should preferably be unique within the enterprise. Although UNIFACE allows the same entity name to be used within different application models, if they are mapped to the same database they will use the same physical table. If it is not possible to prefix entity names with {OWNER_CODE}_ in the application model, then it can be done in the assignment file.

4. The DBMS type should always be declared as DEFAULT within the application model. This can be assigned to a particular database type at run time by an entry in the .ASN file.

2.2.3 Entity (non-database, defined locally in component)

1. To avoid confusion with 'normal' entities these should be named 'DUMMY' or 'DUMMYn'.

2. The non-database property will disable the **retrieve/o** instruction. If it is necessary to use this instruction (eg: in a Select/Remove type of user interface) then this object should be defined within the application model, but with the Read/Write triggers made empty.

3. If the same non-database entity is referenced in several components (eg: ACTION_BAR, NAVIGATION_BAR) it may be worthwhile to define it within the application model.

2.2.4 Entity (non-database, defined in application model)

1. This facility should be used for non-database entities that are commonly used throughout the application. The following "dummy" entities are used routinely within these standards:-
   - ACTION_BAR - to contain all action buttons.
   - NAVIGATION_BAR - to contain all navigation buttons.
   - RETRIEVE_PROFILE - to enter selection criteria at the top of List forms.
   - COLUMN_BAR - to contain label buttons above each row in a multi-row form.

2. Because these entities only exist within the repository, there is no need for uniqueness within the enterprise.

3. The non-database property will disable the **retrieve/o** instruction. If it is necessary to use this instruction (eg: in a Select/Remove type of user interface) then change this property, but empty the Read/Write triggers.

2.2.5 Subtype

1. For subtype naming, DBMS length restrictions do not need to be taken into consideration because subtypes are mapped to the DBMS table or file that corresponds with the entity.

2. If a subtype is created just to allow the separate entry of profile values to be used in a retrieve, consider the use of a dummy entity called RETRIEVE_PROFILE instead.

3. The subtype name should preferably be the supertype name with a numbered suffix (_S01, _S02). This helps to immediately identify the table as a subtype, and identifies its supertype.

4. If more meaningful values for the suffix are required they should not be too long, eg:
   - _PREV (for the previous entry in a sequence)
   - _NEXT (for the next entry in a sequence)
   - _SNR (for the senior entry in a hierarchy)
   - _JNR (for the junior entry in a hierarchy)

5. Other values that may be used include the following:
   - LOCATION_FROM and LOCATION_TO (different relationships to LOCATION)
   - INVOICE_PAID (subset of INVOICE)
   - MANAGER (type of EMPLOYEE)

6. Any conditions that are specific to the subtype can be added to the entity triggers. Note that it is not possible to change any of the field definitions or field triggers for a subtype.

2.2.6 Database View

1. Database Views can only be employed when supported by the underlying database. The use of Views may restrict the range of database engines that could potentially be used to support the application.

2. UNIFACE itself does not recognise whether an entity is used as a table or a view.

3. It is recommended to store views in a separate database model, eg: {owner_code}_VIEW. Care should be taken not to use this application model when generating Create Table scripts.

4. To avoid confusion with ‘real’ entity names, and to avoid conflicting with identical names in other models, it is recommended that a prefix be used, either “V” or “V_”.

2.2.7 Field

1. Although UNIFACE allows field names of up to 32 characters it is recommended that no more than 14 be used (or should be unique within the first 14 characters). This will allow the field name to be used as the message id for retrieving text from the message file.

2. Although field names must be unique within an entity, it is possible to use the same name on multiple entities even if there is no relationship between them. This practice should be avoided as a field's name is used to identify other associated objects (eg: label text, help text), so a field name should not be re-used unless it can share the same context. For example, entities named *Account*, *Invoice* and *Invoice_Line* may each require a field to hold a status value, but these should be given separate names such as *Acc_Status*, *Inv_Status* and *Inv_Ln_Status*.

2.2.8 Primary Key Field

1. It is recommended that the primary key of an entity should be constructed from the entity name with a suffix of "_ID". This makes it easy to identify the primary key in a long list of field names.

2. Avoid using generic names for all primary keys. If the primary key of entity 'A' is 'ID.A', and the primary key of entity 'B' is 'ID.B' and 'A' is linked to 'B' in a one-to-many relationship then 'ID.A' has to be converted to 'A_ID.B' before it can be used as the foreign key on 'B'. This results in the situation were two fields ('ID.A' and 'ID.B') have the same name but different contexts, and also were two fields ('ID.A' and 'A_ID.B') have different names but the same context. This could cause problems when manipulating data in associative lists.

2.2.9 Technical Keys

1. Avoid the unnecessary use of technical keys. If a table already contains a satisfactory unique identifier there is no need to create another one.

2.2.10 Foreign Key Field

1. If the name of the foreign key is identical to the name of the corresponding primary key, and the primary key is constructed from "<entityname>_ID", this makes it easier in a field list to identify which is a foreign key, and which to which entity is it related.

2. Where multiple relationships to the same entity are implemented, using subtypes, use the name of the subtypes for the fields that act as foreign keys. For example, the *Person* entity contains the fields *Country_Of_Birth* and *Country_Of_Residence* to define two relationships between *Country* and *Person*. Alternatively you may add a meaningful suffix to the field name to identify the difference, as in *Location_Id_From* and *Location_Id_To*, or *Part_Id_Snr* and *Part_Id_Jnr*.

2.2.11 Candidate Key Field

1. It is recommended that any candidate keys should include a suffix of "_ID". This makes it easy to identify them as indexed paths.

2.2.12 Relationships

1. Relationship names are only required when creating referential integrity checks. This name is used when SQL script files are generated by the Create Script utility.

2. It is recommended that the name be a combination of the first 10 characters of the ONE entity followed by the first 10 characters of the MANY entity, separated by an underscore.

2.3 APPLICATION LIBRARY

2.3.1 Library Name

1. Each application should have its own library. This should be defined as the library in the properties of each component within the application. It should also be defined as the setting for **$variation** in the application's INIT_PROC.

2. It is possible to change the library that is referenced at run time by altering the contents of **$variation**. This would present a housekeeping problem in that it would require separate copies of the same objects to be maintained in different libraries, therefore it is recommended that each application use only one library.

3. The USYS library should be reserved for global objects (messages, glyphs, panels, menus) that are not specific to any application. Standard entries are contained in:
   - *Appendix J: Global Messages*
   - *Appendix L: Menu Bars*
   - *Appendix M: Panels*

4. The SYSTEM_LIBRARY should be reserved for global objects (procs, include procs, variables, constants) that are not specific to any applications. Standard entries are documented in:
   - *Appendix G: Global Procedures*
   - *Appendix H: Global Variables*
   - *Appendix N: Global Constants*

5. The STD library should be reserved for include. Standard entries are documented in:
   - *Appendix O: Include Procs*

2.3.2 Counters

1. Do not use the USYS library for your counters. Do not change the values of counters stored in the USYS library.

2. A UNIFACE counter uses the Proc statements **numgen** and **numset** to create and maintain a counter. These counters are stored in the table UOBJ.TEXT, which is committed through the $UUU path. This is the easiest way to implement generation of keys.

3. The disadvantages of counters are:
   - Copying counters into another run-time environment needs UNIFACE export and import facilities.
   - The counter value is committed immediately through $UUU, therefore it is not possible to commit the counter value and the occurrence with a single commit.
   - Because it is stored in UOBJ.TEXT counters cannot be used in a self-contained service.
   - In a large network where the application is duplicated across multiple application servers it will not be possible to share the same numbering sequence.

4. The use of UNIFACE counters is discouraged in these standards - please refer to section *4.16 Obtaining Next Number in a Sequence* for more details.

2.3.3 Global Constants

1. These are referenced in code by enclosing the name with "<" and ">", as in "<auto_retrieve>". The name of the constant is replaced with the expression when the component is compiled.

2. A local constant with the same name may also be defined within an individual component. In this case the value of the local variation is used in preference to the central definition.

2.3.4 Global Procedures

1. SYSTEM_LIBRARY should be used for system-wide objects (do not use USYS).

2. The standard global procs used within these standards are shown in *Appendix G: Global Procedures*.

3. Application-specific objects should be defined in the individual application library.

4. Self-contained components cannot use global procedures, but alternatives can be defined as Include Procs (described in a later section).

2.3.5 Global Variables

1. SYSTEM_LIBRARY should be used for system-wide objects (do not use USYS).

2. The standard global variables used within these standards are shown in *Appendix H: Global Variables.*

3. Application-specific objects should be defined in the individual application library.

4. In a modal environment global variables are used to pass values from one form to another, but in a non-modal environment there is a different mechanism for the passing of parameters, so the use of global variables should be kept to a minimum.

2.3.6 Glyphs

1. The only glyphs used in these standards are:
   - U_POPUP_BUTTON for the popup button
   - MENULOGO for the default image in the logo screen

2.3.7 Help Text

1. UNIFACE help text is not used with these standards as an alternative method of maintaining and displaying text is available. Please refer to section *4.15 Online Help* for more details.

2.3.8 Include Procs

1. Include Procs are an alternative for global Procs. However, there are some differences:
   - Include Procs are not compiled immediately when defined in the library. They are expanded and compiled during the compilation of those components which reference them.
   - Include Procs are part of the .**frm**, .**srv** or .**rpt** file, while compiled global Procs are saved in the UOBJ.TEXT table or a **.dol** file.

2. Include Procs are referenced using the **#include** statement:
   #include {Library:}Module
   If *Library* is omitted, the library of the current component is used. If no component library is defined, the SYSTEM_LIBRARY is used.

3. A standard set of Include Procs are defined in the STD library for use in self-contained components. These create local procs with the same name as global procs. This is especially useful when the application model contains references to Procs which can only be satisfied with locally-defined equivalents. These included procs have the same names as Global Procs where they perform a similar function, but may contain minor alterations to allow them to run in self-contained components. If any changes are made to these Include Procs then all associated components must be recompiled so that the changes can be incorporated in the compiled components.

4. Additional Include Procs are defined in library STD which can be used instead of keying in commonly occurring blocks of proc code. A typical example is:

```
#include STD:FATAL_ERROR
```

   which can be used instead of

```
if ($procerror)                    ; check for fatal error
  call PROC_ERROR($procerrorcontext)
  $status = <FATAL_ERROR>
endif
if ($status = <FATAL_ERROR>)       ; check for fatal error
  call GET_MESSAGE                 ; display error messages
  return(<FATAL_ERROR>)
endif
```

2.3.9 Menus and Menu Bars

1. Menu bars can be assigned in the Start-up Shell and in forms, but can also be assigned in proc code if required.

2. The menus used within these standards are detailed in *Appendix L: Menus.*

2.3.10 Messages

1. Messages should be obtained with the **$text(id)** function so that the message text can be changed, or foreign language variants introduced, without the need to change any code.

2. Due to length restrictions it may be necessary to use numbers instead of letters in {name}.

3. The message types used within these standards are detailed in *Appendix I: Format for Message File Entries*

4. The standard messages used within these standards are detailed in *Appendix J: Global Messages.*

5. There can be many different messages associated with a particular topic, and it may help locate them if the topic name (eg: START_DATE, END_DATE, QUANTITY) were to be defined in the description field. Use upper case characters as this will make searching easier.

2.3.11 Panels

1. Panels may be referenced in the start-up shell, or in components.

2. Entries defined as Panels may be referenced as Pop-up Menus.

3. The Panel feature is not used within these standards as the relevant options are usually available on command buttons within each component.

4. The standard panels used within theses standards are detailed in *Appendix M: Panels.*

2.3.12 Pop-up Menus

1. Pop-up menus are defined within the application library as Panels.

2. Prior to UNIFACE version 7.2 a pop-up menu could only be referenced in the start-up shell. Separate menus can now be defined for fields, entities and components.

3. The fallback path is **field** -> **entity** -> **component** -> **start-up shell**.

4. The standard panels used within these standards are detailed in *Appendix M: Panels.*

2.4 TEMPLATES

2.4.1 Entity Interface Templates

1. Avoid names starting with the letter “U” to prevent confusion with UNIFACE objects.

2. Entity Interface Templates apply in situations where included entities are used. Because the use of included entities is no longer recommended, Entity Interface Templates are hardly used

2.4.2 Field Interface Templates

1. Standard entries are detailed in *Appendix C: Field Interface Templates.*

2. See Field Template for further comments.

2.4.3 Field Syntax Templates

1. Standard entries are detailed in *Appendix D: Field Syntax Templates.*

2. See Field Template for further comments.

2.4.4 Field Layout Templates

1. Standard entries are detailed in *Appendix E: Field Layout Templates.*

2. See Field Template for further comments.

2.4.5 Field Templates

1. Standard entries are detailed in *Appendix F: Field Templates*

2. Having the possibility of sharing field interface/syntax/layout templates between several field templates may not be practical. If a shared template were changed this would result in a change to all the associated field templates. If it should ever be required to make changes only to those fields which use a particular field template it would be advisable not to share any interface, syntax or layout templates, but to use separate ones for each field template (using the same name).

2.4.6 Component Templates

1. Standard entries are detailed in a separate document:
   - *UNIFACE Development Guidelines, Part 3 - Component Templates.*

2. Do not modify any of the standard templates. Instead make a copy and add an owner code.

2.5 COMPONENTS

2.5.1 Component Name (Form, Service or Report)

1. Each form component name should conform to the pattern **xxS####D** where:
   xxS is the application prefix
   #### is the sequence number
   D is the dialog type from *Appendix K:*

2. Forms that can be grouped together into a 'family' should share the same sequence number. The only difference in the name need be the dialog type.

3. Each service component name should conform to the pattern **xxSZ####** where:
   xxS is the application prefix
   Z is the dialog type (eg: H for Hidden)
   #### is the sequence number

4. Because part of the form name is a meaningless sequence number it is essential that the description/comments fields of form properties be completed to indicate the form's purpose.

2.5.2 Component Instance

1. A component instance is an occurrence of a component that exists at run-time. The design may call for only one copy of a component to be in existence at any one time, or it may allow multiple copies to exist alongside each other. The name will therefore need to be constructed differently depending on which choice is made:
   - For a single copy use the component name, as this is already unique.
   - For multiple copies construct a name which combines elements of the original component name, plus an identifier for each separate copy.

2. Refer also to section *4.7.8.4 Choosing an Instance Name.*

2.5.3 Local Procedures

1. In order to be easily differentiated from global procs all local procs should have an 'LP_' prefix.

2. Although local procs can be defined in almost any trigger, it is advisable to keep them all in the <local proc module> trigger at component level, thus making them easier to find.

3. Components which are derived from component templates may have some local procs defined in other usually redundant triggers. This is to allow local variations to be made in the <local proc modules> trigger without losing the inheritance properties of these modules.

3. DEVELOPMENT STANDARDS

3.1 THE ENVIRONMENT

3.1.1 The Directory Structure

Rather than have all files residing is a single directory regardless of their file type, I prefer to break them down into sub-directories. It looks much neater (in my opinion) and makes them easier to observe and manage. The directory structure should resemble the following:-

- project directory
  - DATA — for application data
    - MENU — data for demonstration MENU system
    - XAMPLE — sample data for XAMPLE system
  - DICT — for development meta dictionary
  - DOL — for Dynamic Object Library
  - URR — for UNIFACE Runtime Repository
  - EXP — for UNIFACE EXPort files
    - MENU — for MENU EXPort files
    - XAMPLE — for XAMPLE EXPort files
  - FRM — for application forms
    - MENU — for MENU forms (optional)
    - XAMPLE — for XAMPLE forms (optional)
  - IMAGES — for image files
  - PRO — for procedure listings (from compilations)
  - TRX — for database conversion files
  - TXT — for application text files
    - MENU — for MENU text files
    - XAMPLE — for XAMPLE text files

Note that in a shared environment some of these directories may exist on a networked drive or a central file server. If the data is held in a single database then the separate sub-directories will be redundant.

The **URR** directory may be combined with the **DOL** directory to have a single place for compiled objects.

The **MENU** sub-directory is used exclusively for the Menu system files.

The **XAMPLE** sub-directory is used exclusively for the dummy application that demonstrates working examples of each of the component templates.

3.1.2 The Initialisation (.INI) File

The Initialisation file defines the appearance and behaviour of the UNIFACE application. It contains numerous settings, among which is the following:-

Accelerator keys	- to define short cuts for menu items
Widgets	- widget properties (standard and custom)
Fonts	- govern the application appearance

Default Widget Settings are shown in *Appendix A:*

Default Fonts are shown in *Appendix B:*

Various settings can be altered while running UNIFACE - from the system menu (activated via the system box in the top left-hand corner of the screen) select the SETUP menu. Please refer to the *Environment Specific Guide*, Chapter 2 *Customisation* for more details.

Although numerous components can be customised according to individual preferences it should be pointed out that certain settings (eg: font sizes, screen sizes) should not be too different from those that will be in use at the client's site, otherwise the appearance of the screens may be adversely effected.

There are three ways that UNIFACE accesses its .INI file, in the following order or priority:-

1) **/ini=<name>** on the command line.

2) **<application>.ini** in the windows directory.

3) **usysXX.ini** in the windows directory

When using the IDF only option (2) is excluded from the search path.

Since the majority of clients will be accessing the software via the use of an application server rather than on local machines, the .INI file will also reside on the application server. In this case the command line for the application will contain the **/ini** command. It is recommended that a version of the .INI file be constructed which will be delivered to the client. This should be used when the application goes through system testing.

3.1.3 The Assignment (.ASN) File

The Assignment file defines the environment for the application. It contains entries for the following:-

- UNIFACE system settings
- Application settings (logical names and values)
- Specifies the language to be used for messages and help text
- Specifies the keyboard and device translation tables to be used
- Defines the location of all files used by the application

There are three ways that UNIFACE accesses its .ASN file, in the following order or priority:-

1) **/asn=<name>** on the command line.

2) **<application>.asn** (or **idf.asn** when using the IDF) in the working directory.

3) **usys.asn** in the directory specified in the .INI USYS section.

Unlike the .INI file processing, UNIFACE will read the **usys.asn** file then overlay any specifications from (1) or (2)

The contents of this file should not need to be varied once created. However, there are some settings that may be useful during system development:-

$search_object **dbms_first**	This determines the order in which the UNIFACE application looks for compiled objects. Possible sources are the dbms file UOBJ.TEXT or a Dynamic Object Library (DOL) file. By setting this option it will not be necessary to recreate the DOL file before testing the application. The live application should use the setting **file_only**.
$search_descriptor **dbms_first**	This determines the order in which the UNIFACE application looks for entity and signature (component) descriptors. Possible sources are the dbms files ULANA.DICT and USYSANA.TEXT, or the UNIFACE Runtime Repository (URR) file. The live application should use the setting **file_only**.
$putmess_logfile=<name.txt>	Copy contents of message frame to a file (useful when debugging).
$transcript_logfile=<name.txt>	To contain the contents of what used to be the transcript window.
$print_assignments	Prints definitions from the assignment files to the message frame. Where there are local and global assignments this will show which is referenced first.

3.1.4 Menu and Security system

A standard menu and security system has been developed to act as a font-end to all systems. This is described fully in a separate document. It consists of the following objects:-

Mnu*.frm	compiled form components
Mnu*.svc	compiled service components
Menu.aps	start-up shell
Idf.asn	assignment file for development
Menu.asn	assignment file for test application
Usys7x.ini	initialisation file with standard settings (widgets, fonts, etc)
STDuobj.dol	compiled objects (for library MENU, SYSTEM_LIBRARY, USYS)
STDudesc.urr	component signatures for menu forms
Field_template.exp	field interface/syntax/layout templates
Component_template.exp	Standard component templates
Initial_values.txt	initial values for the menu database

To install this system please follow these steps:-

1. Copy files into relevant directories.
2. Import all the **.exp** files.
3. Compile all central objects.
4. Adjust the assignment file to identify the location of the MENU database.
5. Create an icon to run the MENU application.
6. Run the application.
7. As the MENU database is currently empty, the first screen will be a file box (refer to function MNU_9010R in the Menu system documentation). Select file INITIAL_VALUES.TXT. After the contents of this file has been loaded the filebox will be shown again. Press cancel.
8. The logon screen will be displayed - enter **AJM** (user id) and **password** (user password). Alternative user id's are **MGR** and **MANAGER**.
9. The first menu screen should be displayed - enjoy!

3.1.5 Component Templates

Instead of building components from scratch it is possible within UNIFACE 7 to create a form, service or report component from a component template. This results in a component which has the basic structure and enough pre-loaded trigger code for it to function in the standard manner, after which it can be customised by including any additional entities, fields or trigger code as is necessary for it to meet its specific requirements.

Part 1 of this manual, which deals with the external look-and-feel of UNIFACE systems developed using these standards, identifies a series of standard dialog types. Each one of these dialog types has a corresponding component template, thus making it much easier for the developer to create a component that corresponds to one of these dialog types.

These component templates are described in a separate document: *UNIFACE Development Guidelines, Part 3 - Component Templates.*

An example system has been created which contains working examples of each of these component templates. This system is called XAMPLE, and is described in a separate document: *UNIFACE Development Guidelines, Part 4 - Xample Application*.

3.2 APPLICATION MODEL

3.2.1 Keys

Every entity must have a Primary Key. This must be constructed using the minimum number of fields on that entity which will uniquely identify each separate occurrence.

It is strongly recommended that only fields of type “string” or “numeric” be used as keys - the use of other data types should be avoided.

If other unique identifiers are available they may be defined as Candidate Keys.

Indexed Keys (non-unique) may be defined for performance reasons, eg: to speed up searches.

Avoid the creation of unnecessary surrogate (technical) keys just to reduce the primary key to a single field. For example, in a one-to-many relationship where CUSTOMER is the “one” entity and CUST_ADDRESS is the “many”, the following possibilities exist for entity CUST_ADDRESS:-

(a)	ADDRESS_ID	primary key, using a value obtained from a central counter or control record
	CUST_ID	foreign key to CUSTOMER
	ADDRESS_TEXT	one or more non-key fields which hold the address data
(b)	CUST_ID	primary key, and foreign key to CUSTOMER
	ADDRESS_NO	primary key, using the next available number for this customer
	ADDRESS_TEXT	

Option (b) has the added advantage that the relationship does not need to be indexed separately as the foreign key field forms the leading portion of the primary key, and can therefore make use of the index that is automatically provided for the primary key. By numbering the addresses sequentially within customer it is also much easier to identify the previous or next addresses in the sequence.

3.2.2 Relationships

All relationships should be defined as CASCADE, with the exception of optional foreign keys that should be defined as NULLIFY. This will allow the function of type DELETE 2 (as defined in Part 1 of this manual) to correctly process all subordinate records.

One-to-many relationships will not be indexed, by default. This may be reviewed for individual relationships should performance become an issue.

3.2.3 Field Labels

All fields within the application model should have a default label defined. This should be in the format **$text(L_<fieldname>)** so that the actual text can be obtained from the message file. It therefore follows that entries must be created in the message file for all of these field labels.

When a field label on a form is associated with a particular field and no specific entry is made in the properties for that label, the label will automatically inherit the default properties as defined within the application model. Thus it will not be necessary to define the contents of any label field unless it deviates from the default.

3.2.4 Field Sequencing

The fields for an entity should be laid out in the following sequence:-

a) the primary key (one or more fields)
b) the U_VERSION field (used only to aid UNIFACE performance, does not contain data)
c) any candidate key(s)
d) any foreign keys
e) other fields/attributes

If primary or candidate keys are comprised of several fields it is advisable that they be contiguous and defined within the entity in the same sequence as the key definition. Some DBMS's may support non-contiguous keys or different sequences, but most do not.

Some file systems will not accept any fields in front of the primary key, or will give reduced performance.

This layout also means that the primary key of any entity can be readily identified from the field list itself.

3.2.5 Long String Fields

If a long string field (eg: to hold comments or notes of unlimited length) is required for an entity it should, for performance reasons, be split from that entity and held on a separate table. This table should therefore only contain the following fields:

a) the primary key of the main entity
b) the U_VERSION field (if being used)
c) the long string field of type C*

Although the ability to have more than one field on an entity with a field type of “C*” is technically supported within UNIFACE, not all DBMS’s support this.

3.2.6 Default Trigger code

The placement of as much "standard" code as possible within the application model is highly desirable. The less code that has to be inserted at the external schema (form) level the better.

However, where certain code should be used in some circumstances (eg: for all online forms) but not in others (eg: for hidden forms) it will be necessary to strike a balance.

- If the code is not defined in the application model then it will have to be manually inserted in those instances where it is required, with the possibility that it could be missed out.

- If the code is defined in the application model then it will have to be manually deleted in those instances where it is not required, otherwise it may perform unnecessary processing.

- A possible alternative option would be to define the code in the application model *as comment lines only*, so that they could be activated merely by removing the semi-colon prefix.

As entities and fields are created default procedure code for their triggers is automatically loaded from the message file. The names of the Message File entries associated with each trigger can be found in the Proc Language Reference manual section 3.2.1.

The defaults are usually obtained from the USYS library using language USA. Local variations can be defined within your application library, but in order to reference them you must update your assignment file as follows:-

a) Include the command **$search_object dbms_first** to cause UNIFACE to search the UOBJ database table instead of the UOBJ.DOL file.

b) Set **$variation** to the name of the application library.

c) Set **$language** to “USA” (although we may use “UK” for our own development, if any other language code is used you must be aware that if any entry with that language code is not found then UNIFACE will default to “USA”, not “UK”).

It will also be necessary to set these values for *default project library* and *default project language* in your developer preferences within the IDF.

When defining new default trigger code it would be better to reference global procedures rather than define blocks of actual code. In this way it would be possible to make global changes without having to recompile any components.

3.2.6.1 Entity Triggers

The following default trigger code is recommended for all entities:-

TRIGGER NAME	CODE	MESSAGE FILE NAME
<add/insert occurrence>	call DISABLE	ADDINSOCC
<help>	call HELP_PROC	GHELPDEF
<leave modified key>	#include STD:LMK_TRIGGER	LEAVEMODKEY
<on error>	call ON_ERROR_E	ONERRORENT
<remove occurrence>	call DISABLE	REMOVEOCC
<validate key>	#include STD:VLDK_TRIGGER	VLDKDEF
<validate occurrence>	#include STD:VLDO_TRIGGER	VLDODEF

The <add occurrence> and <remove occurrence> triggers are disabled as these operations will be controlled by buttons on the action bar.

3.2.6.2 Field Triggers

The following default trigger code is recommended for all fields:-

TRIGGER NAME	CODE	MESSAGE FILE NAME
<on error>	call ON_ERROR_F	ONERRORFLD

3.3 COMPONENTS

3.3.1 Component Properties

The following items will need to be modified for each form:-

Description	optional	Short description of the form
Title	optional	Whatever is defined here will be overwritten at run-time by an entry from the message file
Library	required	Set to name of application library
Menu Bar	optional	Set as appropriate
Popup Menu	optional	

The correct form properties should have been set in the relevant component template (see above). The following values should not need changing:-

Behaviour	Normal	Allows database updates
	Limited	Read-only
	Code List	Not used
	Record	Not used
	Help	Not used (except in HELP forms)
	Menu	Not used
Keep Data in Memory	Off	Only turned ON for performance reasons
Drop Component from Memory	On	Only turned OFF for performance reasons
Self Contained	Off	Only applicable for Services and Reports
Clear Area	Off	Only sensible in character mode
Border	Off	Only sensible in character mode
Panel	blank	Not used - all actions should be defined as buttons within the Action Bar
Communications Default	Synchronous	

The Comments area should contain a block of text similar to the following:-

```
Function:        MNU_0001 - LOGON screen

Description:     This is the first screen into the system..

Author:          Tony Marston

Date Written     21-06-99

Current Version: 1.0.0

Update History:

Date        Updated By          Details
```

3.3.2 Window Properties

The form's Window Properties should not need changing from those that were set it the component template. They should be defined as follows:-

Window Type	Defines the behaviour of the form	Options are:- Normal (default) Primary Secondary Tab Page
Modality & Attachment	Defines how the form is to be run	Options are:- Modal, Attached Non-Modal, Attached Non-Modal, Detached
Caption	Enables the title bar at the top of the form	ON (default)
System Menu	Enables the system menu for the form	ON (default)
Iconize	Enables an iconize button in the form caption	ON (default)
Maximise	Enables a maximise button in the form caption	ON (default)
Close	Enables the ability to exit from the form via the standard keystrokes or exit procedure of the GUI	ON (default)
Resizable	Enables the form to be resized	ON (default)
Overlay Previous Form	Causes the form to appear as an unmoveable form over the previous form	OFF (default)
Hide Previous Form	Makes all other open forms disappear from view (but not dropped) when this form is run	OFF for popups OFF for hidden forms OFF for help ON for others

3.3.3 Component Triggers

The following triggers may be populated automatically from the component template:-.

Execute	This is the default entry point for the module.
Clear	Empty, unless a CLEAR button is available.
Retrieve	Empty, unless a RETRIEVE button is available.
Retrieve Sequential	May contain local procs inherited from the component template – by being defined in a separate trigger they will not lose their inheritance unless the trigger is modified.
Accept	Call OK_PROC or CLOSE_PROC if no OK button is available.
Quit	Call QUIT_PROC or CLOSE_PROC if no CANCEL button is available.
Async Interrupt	May contain default code to handle messages from a child instance.
Local Proc Modules	May contain default code, but can be modified as necessary. Note that if any modification is made then the ability to inherit current values from the component template will be lost.
Operations	Usually contains a default INIT operation. Other operations can be added as required. Note that an operation will retain inheritance from the component template unless that operation has been modified locally, regardless of any changes to other operations.

The following triggers will usually be empty (disabled):-

<store>	Database updates are only performed when the <accept> trigger is invoked.
<erase>	Records can only be deleted if there is a specific transaction or function key for that purpose.
<print>	Printing from the screen is disabled.
<user key>	Not usually required
<pulldown>	Not usually required
Form Gets Focus	Not usually required
Form Loses Focus	Not usually required

NOTE: references to a button (eg: OK, CANCEL, CLEAR, RETRIEVE) indicate that a specific button is available within the form's Action Bar (see below). If a trigger has not been disabled then a corresponding button should be available in the form's Action Bar. The user is therefore able to fire the trigger either by pressing the relevant button, or by using the trigger's particular sequence of key strokes (see the section on short-cut keys in Part 1 of this document).

3.3.4 Local Procedures

Although the component template may contain some default procs in other triggers, it is good practice to keep all others in the <local proc module> trigger.

Note that local procedures have an “LP_” prefix to avoid confusion with global procedures, which have no prefix, and which may be found in one of several application libraries.

As this trigger may contain many different entries the procedure name should be in uppercase, and should be included on the **end** statement. There should be a separator between one procedure and the next (eg: a line of hyphens or asterisks). The procedure should also contain some brief comments that outline its function. This can be represented as follows:-

```
entry LP_INITIALISE    ; local initialisation procedure

<full.<retrieve_profile>>        = $text(B_FULL_PROFILE)

end  LP_INITIALISE
;-------------------------------------------------------------------------------------------------------
entry LP_VALIDATE              ; check that all entries exist and are valid
.............
.............
.............
end LP_VALIDATE
```

In cases where a proc name contains several words which try to identify its meaning, and these words have to be strung together without a separator due to size limitations, it may make it more readable to use uppercase for the first letter of each word, as in *lpFindFirstEntry*.

3.3.5 Local Constants

Local constants have two parts, a *NAME* and an *EXPRESSION*. The constant can be referenced anywhere in proc code by using *NAME* surrounded by "<" and ">", as in "<name>". When the component is compiled all instances of <name> in the resulting object code will be replaced by the value of the expression associated with that name. This will only be visible in the proc listing.

This facility is used in local procs that are inherited from component templates. Where a proc requires a name or value, if this value is hard-coded within the proc then that proc cannot be altered without cutting off all inheritance from the component template. However, if the proc contains references to local constants then the value of *EXPRESSION* can be altered within the local constants screen without changing any code within the proc, and without losing any inheritance.

For example, the INIT operation in most forms contains the following:

```
operation INIT

$form_version$          = "<form_version>"
….

end INIT
;=================================================================
```

When the component is compiled the reference to **<form_version>** will be replaced with the current expression for the constant with that name.

3.3.6 Button Bars

Each form may contain sets of pushbuttons that are arranged as follows:-

⇒ An Action Bar running horizontally across the bottom of the form.

⇒ A Navigation Bar running vertically down the right-hand side of the form.

⇒ A Column Bar running horizontally across a multi-row display.

For convenience all of these button bars should be defined in the application model as dummy (ie: non-database) entities. The most commonly used buttons should then be defined as dummy fields within these dummy entities.

The name of the button should equate directly with the function performed by that button. This then makes it possible to search through the IDF source in order to identify all those forms that perform that function (ie: contain that button).

The properties of the buttons should be set as follows:-

dimensions	width=11, depth=2
data type	string
widget type	fCommandButton

	template	**shorthand**
interface definition	PUSHBUTTON	C22
syntax definition	PUSHBUTTON	NED
layout definition	PUSHBUTTON	CTR,NAV (see note)

characteristics	BOILERPLATE or CONTROL	(see note)

1. **NAV** will prevent the button from changing colour to the value specified in **$active_field** whenever the field has focus.
2. **BOILERPLATE** will activate the <leave field> trigger of the current field, and any associated entity triggers, if necessary. **CONTROL** will not activate these triggers and should therefore be reserved for the CANCEL, CLOSE, DELETE and CLEAR buttons.

Each button will contain a label in the form of a text string that will be retrieved from the message file. The label cannot be loaded using the INITIAL VALUE option within field properties, therefore must be the subject of a specific assignment in proc code.

The button's **<detail>** trigger must contain the code to be executed when the button is activated.

All buttons on the ACTION and NAVIGATION bars should be included in the form's prompt sequence - ie: it should be possible for the user to gain access to any button by means of the TAB or BACKTAB key.

As the cursor is positioned on each button a hint message should be displayed in the message line in order to give the user more information as to the purpose of the button. The hint text should be taken from the message file using an identity constructed from the button name prefixed by "H_". The code to display this text should be inserted into the button's **<field gets focus>** trigger, as in the following example:-

```
message/hint $text("H_%%$fieldname")
```

3.3.6.1 Action Bar

A button on the Action Bar will perform specific processing on the data within the current form.

The name of the button should equate directly with the function to be performed by that button (eg: OK, CANCEL, CLEAR, and RETRIEVE).

The button label should be loaded from the message file. The identity of the message file entry should be the button name prefixed with "B_". The code to load the button labels would therefore be similar to the following:-

```
ok.action_bar       = $text(B_ok)
cancel.action_bar   = $text(B_cancel)
```

NOTE: The loading of labels for entity ACTION_BAR is automatically performed by global proc ACT_BUTTONS, which is invoked by the INIT operation.

Where an action button equates directly with a form-level trigger it must be possible for the user to perform that action either by pressing the button or by using the trigger's particular short-cut key (as mentioned in Part 1 of this document). To ensure that both methods perform exactly the same processing it is advised that the necessary code be placed in the form-level trigger, and for the button to pass control to the trigger by using the **macro "^TRIGGER"** statement.

3.3.6.2 Navigation Bar

A button on the Navigation Bar will invoke another (child) form in order to perform additional processing, usually associated with the object that is being displayed on the current form.

The name of the button should equate directly with the name of the form that is run when that button is pressed. (eg: MNU_0030L, MNU_0030C, etc).

The button label should be loaded from the message file. The identity of the message file entry should be the button name prefixed with "B_". The code to load the button labels would therefore be similar to the following:-

```
MNU_0030L.navigation_bar        = $text(B_MNU_0030L)
MNU_0030C.navigation_bar        = $text(B_MNU_0030C)
```

NOTE: The loading of labels for entity NAVIGATION_BAR is automatically performed by global proc NAV_BUTTONS, which is invoked by the INIT operation.

An example of the proc code that is required to activate a child component is detailed in section *4.7.8 Invoking a UNIFACE component.* This code may have to be modified under the following circumstances:

⇒ If a non-empty occurrence on the current screen has to be selected before the child form can operate then code similar to the following will be required:

```
If ($empty(<entity>)
   Message $text(M_90009)          ; nothing retrieved yet
   Return(-1)
endif
```

⇒ If any changes to database values have been made on the current form these changes must be stored on the database before control is passed to the child form so that it has access to the current values, not the previous values.

If the current form is of type LIST and has the potential for dealing with stepped hitlists then please refer to section *4.18 Hitlist Processing* for more details.

If the child form modifies the database, and these modifications need to be communicated back to the parent so that they may be incorporated into the parent's display, then please refer to section *4.8.2.2 Receiving a message* for more details.

3.3.6.3 Column Bar

The Column Bar is designed to be used in those forms which contain multiple occurrences arranged in rows, with a label above each column. A column button replaces an ordinary label as it allows processing to be performed should the user click on it with the mouse.

When a column button is pressed it will cause the hitlist of retrieved occurrences to be sorted according to the value contained in that column. It will compare the value from the first and last occurrences in order to determine if the current sequence is ascending or descending, then sort the hitlist in the other sequence.

Each button on the Column Bar should have exactly the same name as the field whose data is being displayed in the corresponding column. Once a button has been created its properties should be set by applying the field template COLUMN_BUTTON. This contains all the default definitions, including code in the <detail> trigger to perform the sort.

The INIT operation for the form will contain a call to the global proc COL_BUTTONS which will load a label into each button. This will be retrieved from the message file using an identity constructed from the button name plus a prefix of “L_”. This should retrieve the same text as the label for the field of the same name.

There are two ways to override this action: -

a) Define an alternative button label in the ‘Initial value’ part of the button’s properties. This will NOT be overwritten by the COL_BUTTONS procedure.
b) Place code in the local proc LP_INITIALISE to replace the value inserted by COL_BUTTONS. This should be similar to the following:-

```
person_name.column_bar      = $text(L_PERSON_NAME)
```

The <detail> trigger should contain the following code from the field template: -

```
call LP_COLUMN_BUTTON($fieldname)
```

The local proc LP_COLUMN_BUTTON will be inherited from the component template, and will contain the code which will perform the sort.

If any field is not actually on the entity being sorted, the button’s <detail> trigger should be changed to something similar to the following: -

```
call LP_COLUMN_BUTTON(“%%$fieldname.ENTITY”)
```

3.3.7 Widget Types

When painting objects on forms the following Widget Types should be used:-

fEditBox	For all editable non-numeric fields - supports proportional fonts, does not support the WIDTH layout model. Multi-line fields containing carriage-returns will not be supported unless the MULTILINE and WORDWRAP options are set ON, or the data type is set to Special String.
fNoEditBox	For all non-editable fields - similar to EditBox, but has different visual presentation
fEditNumber	For all editable numeric fields
fNoEditNumber	For all non-editable numeric fields
Unifield	Does not support proportional fonts, supports the WIDTH layout model, MULTILINE and WORDWRAP are ON by default
fListBox	For list boxes
fDropDownList	For dropdown lists
fDropNoEdit	For dropdown lists which are display only
fComboBox	For Combo boxes (dropdown lists with the option of user input)
Dynalabel	For dynamic labels
fCheckBox	For checkboxes (fields of type BOOLEAN)
fCommandButton	For pushbuttons or command buttons
fColumnbutton	For column labels on the Column Bar
fMenuButton	Used by the menu system for the menu options
fRadioGroup	For radio buttons
fSpinButton	For spin buttons on numeric fields
Label	For all screen labels (no text to be hard-coded)

Please refer to *Appendix A* for the default settings for these widgets.

The “f” prefix on widget names signifies a custom version that should be used in preference to the standard version provided by UNIFACE. This means that the widget properties used within an application are kept separate from any default definitions that are provided by UNIFACE, which could possibly be changed with a future release of the product.

The custom widgets can be made to replace the standard widgets in the form painter’s tool palette by means of the following entries in the **.INI** file:

```
[gfp]
widgets=fcheckbox,fradiogroup,fcommandbutton,fdropdownlist,fspinbutton
```

4. CODING STANDARDS

Coding standards help to ensure the maintainability of the code through its clarity, and allow for easy integration of different information systems. A standard approach shortens the development process: developers know immediately what to do, and many issues have standard solutions.

4.1 PROC LAYOUT STANDARDS

4.1.1 Structure of Proc Modules

Do not make proc modules too large, instead create a series of small, single-purpose modules. Give each module a meaningful name, preferably something which describes its purpose.

Example:

```
if (option = 1)
        call LP_CALCULATE_1
elseif (option = 2)
        call LP_CALCULATE_2
else
        message $text(M_9001)
        return(-1)
endif
……
;=====================================================================
entry LP_CALCULATE_1
        …. ; lots of code
end LP_CALCULATE_1
;=====================================================================
entry LP_CALCULATE_2
        …. ; lots of code
end LP_CALCULATE_2
;=====================================================================
```

4.1.2 Uppercase, lowercase, mixed case, appropriate case

As a general rule all proc coding should be entered in lower case. THE USE OF UPPER CASE IS CONSIDERED TO BE SHOUTING, and should only be used when something needs to be brought to the reader's attention. Other people may have different opinions, but when I'm looking through code I like to see the following highlighted with the use of uppercase:

- Where a module starts.
- Where a module finishes.
- Where a module branches to another module (eg: with **call**, **run**, or **activate**).

Use the appropriate case for object types whose interpretation depends on the correct use of mixed upper and lower case, such as:

- File names
- Operating system commands
- Specific or required string values

In cases where a proc name contains several words which try to identify its meaning, and these words have to be strung together without a separator due to size limitations, it may make it more readable to use uppercase for the first letter of each word, as in *lpFindFirstEntry*.

4.1.3 Entries

When coding a local proc module (entry) follow these simple guidelines:

- Put the module name in upper case so that it stands out.
- Include a brief description of the module after its name.
- Include the module name on the **end** clause.
- Put a blank line after the **entry** and before the **end**.
- Terminate each module with a distinctive separator, such as a line of "=".

Example:

```
entry LP_CALC_TOTAL        ; calculate total amount
…….
…….
…….

end LP_CALC_TOTAL
;=================================================================
entry LP_SOMETHING_ELSE       ; do something else
…….
…….

end LP_SOMETHING_ELSE
;=================================================================
```

4.1.4 Operations

These can be treated the same way as local proc modules:

```
operation CALC_TOTAL       ; calculate total amount
params
        string   pi_input_string          : IN
        string   po_output_string         : OUT
endparams
…….
…….

end CALC_TOTAL
;=================================================================
```

4.1.5 Parameters

Where proc modules or operations require parameters to be passed these are specified in the *params…endparams* block which must appear immediately after the module/operation name and before the *variables…endvariables* block or the first line of code.

Use one of the following prefixes on parameters so that they can be distinguished from local variables and entity items:

pi_	on IN parameters
po_	on OUT parameters
pio_	on INOUT parameters

4.1.6 Indenting and Spacing

In nested structures identify each separate level by indenting one tab stop more than the previous level. A single tab is easier to use than a multiple of spaces, and the width of each tab can be adjusted by using the UNIFACE ruler. Separate one logical block of code from another with a blank line.

```
if (lv_Componentname = $applname)                    ; being run from start-up shell
        $status = 0                                  ; do nothing
else
        call CREATE_INSTANCE()                       ; check instance name
        if ($status)
                if ($procerror) call PROC_ERROR($procerrorcontext)
                return($status)
        endif
endif

if (lv_Params = "")                                  ; optional
        activate lv_Instance.lv_Operation()          ; without $$PARAMS
        if ($procerror = <UPROCERR_NPARAMETERS>)     ; param mismatch
                activate lv_Instance.lv_Operation(lv_Params) ; try with
        endif
else
        activate lv_Instance.lv_Operation(lv_Params) ; with $$PARAMS
        if ($procerror = <UPROCERR_NPARAMETERS>)     ; param mismatch
                activate lv_Instance.lv_Operation()  ; try without
        endif
endif
if ($procerror)                                      ; could not activate
        call PROC_ERROR($procerrorcontext)
        return(-1)
endif
```

4.1.7 Alignment

Where a group of proc statements are performing similar functions (eg: assigning values) it makes the code more readable if corresponding arguments are aligned, as in:-

```
f1.entityA          = “one”
field2.entityA      = “two”
field_three.entityA = “three”
```

This looks more elegant than:-

```
f1.entityA = “one”
field2.entityA = “two”
field_three.entityA = “three”
```

4.1.8 Line Continuation

Divide proc statements that are larger than 72 characters into multiple lines using the line continuation marker **%** and indent the next line.

```
$$sql_where_clause = “where name = %%$name$%%% %\
   and Date_of_Enrollment < %%$check_date$%%%”
```

4.1.9 Substitution Delimiter

Always end a substitution with **%%%** where confusion is obvious, such as in the following example:

```
use          “The target field is %%TARGET_FIELD%%%.%%$entname$%%%”
rather than  “The target field is %%TARGET_FIELD.%%$entname$”
```

4.1.10 Comments

Some people assume that comments are superfluous as you can surely tell what a module does just by looking at the code! WRONG! Comments are an invaluable aid when the time comes to debug or enhance a module – they explain why a particular line of code exists in a particular place, thus making the maintenance task a lot easier.

Keep proc code short and self-explanatory, thus minimising the need for comments. Avoid having pages of comments, but put short comments where needed, either above or next to the proc, and try to align the comments.

```
; initialisation before check
…..
activate “ABC002H”          ; check business rule BR001
if ($status = 100)          ; BR001 not violated, so continue
      …..                   ; reset what needs to be reset
      return(0)
else                        ; BR001 violated!
      …..                   ; do what needs to be done
      return(-1)            ; stop this processing
endif
```

4.2 STANDARDS FOR USING FIELDS AND VARIABLES

4.2.1 Field Qualification

Always qualify fields (field.entity) in proc coding, except in the following circumstances:-
- When qualifying the field is not allowed according to the syntax of the proc statement.
- In the **order by** clause of a **read** proc instruction.
- The select phrase of the **selectdb** proc instruction.
- The field list of the **compare** proc instruction.
- When defining proc coding in the application model in a field-level trigger with references to that same field or another field in the same entity. This will ensure that the coding will also be valid when used in subtype entities.
- When used in global procs where referring to fields of different entities.

Field qualification is necessary under the following circumstances:
- When in one component several fields in different entities share the same name.
- When a field is referenced in a proc module that contains parameters or local variables with the same name. Unless the field name is fully qualified the value of the parameter or local variable will be used instead of the field value.

4.2.2 General variables ($1 - $99)

These are a relic of early versions of UNIFACE, and should now be avoided wherever possible as their usage and content is not clearly evident from their identities. The current alternatives are:-
- Global variables ($$name)
- Component variables ($name$)
- Local variables
- Parameters (when passing values between objects)

4.2.3 Global variables ($$name)

Their use should be kept to a minimum as:-
- Global variables cannot be used in self-contained components.
- Global variables cannot be used in a partitioned environment to pass values between components on clients and/or different servers.
- The behaviour of modal and non-modal components could cause unexpected results.

When passing values from one component to another it would be better to use the argument list on the **activate** statement.

4.2.4 Component variables ($name$)

Use instead of local variables where a value needs to be referenced by different triggers or procs in the same component.

4.2.5 Local variables

Use these in preference to component, global or general variables.

Use an ‘lv_’ prefix on local variable names so that they can be distinguished from parameter names and entity items.

4.3 STANDARDS FOR USING MESSAGE FILE ENTRIES

4.3.1 Message text

When a message needs to be displayed to the user it is better to obtain it from the message file rather than hard code it inside the proc. This improves flexibility and helps to provide language independent text and labels.

The message line can only contain a single line of text, and will always contain the message that was last issued. The history of past messages can be examined by pressing the  button at the end of the message line.

Messages of more-than-average importance can be brought to the user's attention by attaching one of the **/error/warning/info** switches to the **message** command. This will cause the message to be displayed in a special dialog box rather than in the usual message line. This dialog box will not be cleared until the 

button (the only available button) is pressed. The message will be logged in the message history.

4.3.2 Hint text

If the <field gets focus> trigger for a field contains a **message/hint** instruction this will cause a message to be displayed to the user as soon as that field is given focus. The **/hint** switch causes the message to be excluded from the message log.

This feature is especially useful on command buttons as it gives the user more information than is usually contained within the button text. Note that the PUSHBUTTON field template contains the following default code in the <field gets focus> trigger :

```
message/hint $text("H_%%$fieldname")
```

4.3.3 Help text

Although it is possible to define help text in the message file, and to give the user the ability to display this help text within the online session, this suffers from the disadvantage that the help text cannot be modified without re-creating the .DOL file. In these standards I find it much easier to keep all help text in the application database. With the aid of a simple maintenance screen the user is therefore able to customise all help text with the greatest of ease. This is fully documented in a later section.

4.3.4 Field Labels

All fields should have their labels defined in the message file using the id of "L_" followed by the field name. The value **"$text(L_<*fieldname*>)**" can then defined in the application model, thus making this the default label for any field which appears in a form component.

Note that message file id's cannot be larger than 16 characters. To allow for the "L_" prefix this means that field names should be unique within the first 14 characters.

4.3.5 Button Labels

Buttons should have their labels defined in the message file with an id of “B_” followed by the button (field) name. This will allow button labels to be loaded with the standard statement :

```
fieldname.button_bar = $text(“B_%%fieldname”)
```

4.3.6 Questions

For questions which offer a variety of responses the **askmess** command should be used, as in the following example:-

```
$1 = $text(B_ABANDON)
$2 = $text(B_RETRY)
askmess/question $text(Q_00031),"%%$1,%%$2"   ; abandon or retry ?
if ($status = 1)                              ; abandon
        .............
else                                          ; retry
        .............
endif
```

Note that the text for the replies as well as the question is obtained from the message file.

Messages processed by the **askmess** command are not logged in the message history.

The switch on the **message** and **askmess** command will determine which of the following symbols is displayed in the dialog box to the left of the text:-

 /info  /warning  /error  /question

It is possible to obtain all the replies from a single entry in the message file, thus replacing **“%%$1,%%$2”** with **$text(R_00031)**. However, this makes the number of replies and their associated **$status** values not immediately evident when examining the proc code. It would be possible to alter the number of replies and their sequence within the message file entry without making the corresponding change in the proc code, which could cause confusion.

4.3.7 Form Messages

There may be occasions when it is required to have a message appear inside the body of a form. If the text is fixed (eg: inside a form of type FRONTEND) then the message id can be constructed as “M_<formname>”. If the message is variable depending on circumstances that are determined at run time, then a message id of the format “M_nnnnn” may be used.

4.4 USE OF LIBRARIES AND GLOBAL OBJECTS

4.4.1 General

The use of different libraries for enterprise-wide and application-specific global objects can be very confusing, therefore the following simple rules should be applied:-

a) Use SYSTEM_LIBRARY (NOT USYS) for enterprise-wide global Procs and Variables.
b) Use USYS for enterprise-wide global objects *other than* global Procs and Variables.
c) Use a specific Proc library (eg: named after the application) for all application-specific global procs and variables. Define this library name in each component's properties.
d) Use a specific globals library (eg: named after the application) for other application-specific global objects. This library name should be assigned to **$variation** by proc code within each component, and should also be defined in the **library** field for the component's definition within the Menu database.

Generally speaking items c) and d) above can be covered by a single application library.

Note that global objects cannot be used in self-contained components. This is because services and reports could be deployed on different servers and therefore may not be guaranteed access to the same set of global objects in the **.dol** file as used on the client machine.

4.4.2 Using Include Procs with Self-Contained Components

Self-contained components cannot reference global procs, therefore all proc references must be satisfied by local procs. However, instead of having the entire contents of each proc defined locally within the component it is possible to use the **#INCLUDE** statement to copy the contents from a central library at compile time. The command syntax is as follows:

```
#include <library name>:<proc name>
```

Note the following:
- <proc name> must be defined as an entry in the INCLUDE PROCS area of <library name>.
- The contents of <proc name> are included in the component at compile time and referenced locally at run time.
- If any include proc is modified all relevant components must be re-compiled before the modifications can be referenced at run time.

4.5 PROC CONSTRUCT STANDARDS

4.5.1 If-Else-Endif constructs

When using conditional statements each part should be on a separate line and indented, eg:

```
if ($status = 0)
        call OK_PROC
else
        call ERROR_PROC
endif
```

Multi-level IF statements are allowed, but should be avoided if they become too complex. Statements that are at the same level should be indented by the same amount to make them more readable, eg:

```
if ($status = 0)
        if (<condition1>)
                call PROCESS1
        else
                call PROCESS2
        endif
else
        call ERROR_PROC
endif
```

Single-line IF statements should be avoided, unless they are very simple, eg:

```
if (!$fieldendmod) done
```

```
if ($status) return(-1)
```

A sequence of IF statements can be specified using ELSEIF instead of an IF at the next level, eg:

```
if (<condition1>)
        <statement1>
elseif (<condition2>)
        <statement2>
…
elseif (<condition99>)
        <statement99>
else
        <nothing qualifies>
endif
```

Note that this can be better achieved using the **selectcase** command.

4.5.2 Selectcase-Endselectcase constructs

This is the most efficient way of defining and executing code where only one condition out of a series of conditions can qualify, eg:

```
selectcase $1
case “”
        message “$1 is empty”
case “ABC”
        message “$1 is ABC”
…
case “xyz”
        message “$1 is xyz”
elsecase
        message “$1 has an unexpected value”
endselectcase
```

Note the use of the final ELSECASE which traps any condition which has not been specifically identified. It is strongly recommended that this feature be used because if there is an unexpected value it can be trapped here rather than allowing it to flow through the system where it may cause untold damage.

4.5.3 While-Endwhile constructs

Assign variables to control the **while/endwhile** loop immediately before the start of the proc instruction, and do not use one-liners.

```
$continue$ = “T”
while ($continue$)
        call XP_NEXT_ACTION
endwhile
```

It is possible to use the **break** instruction to terminate the loop without using an additional variable:

```
setocc “entity”,1
while ($dbocc(entity))
        call PROCESS_OCC                    ; process this occurrence
        setocc “entity”,$curocc(entity)+1   ; step to next occurrence
        if ($status < 1) break              ; none left – stop
endwhile
```

4.5.4 Repeat-Until constructs

Assign variables to control the **repeat/until** loop immediately before the start of the proc instruction, and do not use one-liners.

```
$count$ = 0
repeat
        ……..
        $count$ = $count$ + 1
until ($count$ = 100)
```

4.5.5 Boolean

Never test a boolean field by testing for the exact value since the **true** value of a boolean field may be “T”, “Y” or 1 depending on the platform and the DBMS in use.

PREFERRED	TO BE AVOIDED

if ($fieldmod (field.entity))
…
endif
	if ($fieldmod (field.entity) = “T”)
	…
	endif

$still_to_do$ = “T”
if ($still_to_do$)
…
endif
	$still_to_do$ = “T”
	if ($still_to_do$ = “T”)
	…
	endif

4.6 STANDARDS FOR USING TRIGGERS

Most triggers at all levels should be used as intended for normal UNIFACE functionality. This section gives standards for certain triggers in order to obtain special results or to avoid conflicts.

4.6.1 Component Template inheritance

In order to maintain maximum inheritance functionality from component templates use what would otherwise be an empty trigger for template-specific procs. These triggers should never be changed in those components that are derived from it, otherwise all inheritance capabilities with the component template will be lost. Examples of such triggers are: <retrieve sequential> in form components, or the <quit> trigger in services. The <local proc modules> trigger should only be used for component-specific procs, and not template-specific procs.

Operations must be located in the <operations> trigger. To maintain maximum inheritance functionality, consider calls to local proc modules, as in the following example:

```
operation PROCESS
params
        …..
endparams
; code inherited from component template
….
….
call LP_PROCESS

end PROCESS
```

4.6.2 Trigger layout

Triggers that are inherited, either from the application model or from component templates, should be prefixed with a block of comments indicating whether the contents can or cannot be modified. If any modifications are made within a component then this comment block should be modified to indicate that fact.

```
; Inherited Code
; All the code in this trigger is inherited – DO NOT CHANGE
```

4.6.3 Local Proc Modules trigger

Although it is possible to define local procedures in any trigger, it is advisable to centralise them so that they can be found easily.

4.7 STANDARDS FOR INVOKING OTHER OBJECTS

4.7.1 Setting $status

Do not use **$status** in an assignment to set the return value; use only the **return** or **exit** proc instruction.

Example:

use:	**return(3)**
rather than:	**$status = 3** **done**

4.7.2 Argument with Return and Exit

Always use an argument with a **return** or **exit** proc instruction. Use explicit values or constants as the argument as much as possible. Only return **$status** when you can be sure that the return value will be checked in the calling object.

Example:

Use:	**return(0)**
rather than:	**return**
use:	return(-1) **exit(<OK>)**
rather than:	**return($status)** **exit($status)**

4.7.3 Checking return values of Proc instructions

- Check the most likely condition first.
- Check the return status of every component-level I/O proc instruction.
- When inserting proc coding in the <read> trigger after a **read** instruction, first check for a negative return value. If the **read** instruction returns a negative value then any remaining processing in that trigger should be abandoned.

Example:

```
read
if ($status < 0) done
$total_amt$ = $total_amt$ + amount
```

4.7.4 Standard return values for $status

Every UNIFACE object (component, operation, global or local proc module) or 3GL routine that is invoked should set the return value (**$status**) according to a predefined standard, eg:

$status	Result	Description
0	Success	Output parameters are set as expected, and processing can continue.
>0	Success with condition	A special condition applies (warning or info situation). The return value (**$status**) determines what can be expected as output parameters; processing can continue.
<0	Failure	Indicates failure (error situation). The return value (**$status**) determines what the problem is; output parameters are not set up properly and processing cannot continue in the normal way. There may also be a non-zero value in **$procerror**.

4.7.5 The use of $PROCERROR

A new function was introduced with version 7.2.01 of UNIFACE to help differentiate between an error being detected inside an object by proc code, and a failure by UNIFACE to activate the object. This function ($PROCERROR) is set to zero only when the object (proc module or function) has been successfully activated, and is only set to non-zero if the activation or call fails.

This means that immediately after any proc statement which attempts to activate another object the contents of **$procerror** should be examined before the contents of **$status**. If **$procerror** has been set to a non-zero value a description of the error will be contained in an additional function called **$procerrorcontext**. This is a string field (an associative list) whose contents can be appended to the message frame by means of a global proc called **PROC_ERROR**.

Example:

```
Instruction argumentlist
if ($procerror)
        call PROC_ERROR($procerrorcontext)
        return(-1)
endif
if ($status)
        ……
endif
```

The result of **call PROC_ERROR($procerrorcontext)** would be something similar to the following being added to the message frame:-

```
ERROR=-1122
MNEM=<UPROCERR_NARGUMENTS>
COMPONENT=X_LIST1
PROCNAME=LP_RETRIEVE
TRIGGER=RETR
LINE=11
DESCRIPTION=Wrong number of arguments
```

4.7.6 Testing for fatal errors

When using a CALL or ACTIVATE statement it is good practice to test for a fatal error before continuing. A fatal error is categorised as one which results in control not being passed to the specified object (module or component), thus the return value is set by UNIFACE itself rather than the object. This can be caused by such circumstances as:

- object not found
- mismatch in the number of parameters
- a fatal error was detected in the called object itself

This can be handled with the following code.

```
if ($procerror)                    ; check for fatal error
  call PROC_ERROR($procerrorcontext)
  $status = <FATAL_ERROR>
endif
if ($status = <FATAL_ERROR>)       ; check for fatal error
  call GET_MESSAGE                 ; display error messages
  return(<FATAL_ERROR>)
endif
```

Note that this standard block of code can inserted by using the following:

```
#include STD:FATAL_ERROR
```

It is good practice to use the **#include** proc as any future changes to the code can be automatically inherited by the components the next time they are compiled.

4.7.7 Error Handling with Self-Contained Services

The standard method for dealing with errors (validation or otherwise) in forms is as follows:

```
if (<condition>)
        message $text(<MessageId>)
        return(-1)
endif
```

This obtains the text for the specified message from the message file (substituting any values as necessary) before displaying it in the message line, and terminates the current procedure.

This method cannot be continued in self-contained services as access to the message file (part of the DOL file) is denied. The message id must therefore be passed back to current form as this does have access to the DOL file and can therefore obtain the message text. Any values which must be inserted in the body of the message must also be passed back so that they are available when the **message** command is invoked. The procedure for dealing with messages in services is now performed in two distinct stages:

4.7.7.1 Recording a Message

All messages will be written to a special component known as the Message Object. This will hold all messages until instructed to pass them back to the current form. Messages can be written to the Message Object by means of the following:

```
call SET_ERROR (“MessageId”)
return(-1)
```

There are variations of this proc called SET_FATAL, SET_INFO and SET_WARNING for different categories of message. Normal validation failures should be treated as ERRORS while the FATAL status is reserved for catastrophic conditions (eg: store errors). INFO and WARNING should be self-explanatory.

Values for insertion/substitution in the body of the message must be included with the message id but separated with the ‘<GOLD>semi-colon’ delimiter, as follows:

```
Call SET_ERROR(“M_90024;1=param1;2=param2;$prompt=<fieldname>)
```

The insertion points should be indicated in the body of the message using **%%$1**, **%%$2** etc, up to **%%$5**. If **$prompt** is specified the cursor will be positioned on that field when the message is eventually displayed.

4.7.7.2 Displaying a Message

The procedure which activates a service must now follow this with a call to the GET_MESSAGE proc which will obtain any outstanding messages from the Message Object, delete them and display them. The count of messages of type “F” (Fatal) and “E” (Error) will be returned in **$status** and can therefore be used to detect that an error message was generated in earlier processing.

```
call GET_MESSAGE
if ($status) return(-1)
```

4.7.8 Invoking a UNIFACE component

4.7.8.1 A Form component

While within an existing form the user may elect to run another form by selecting a command button or menu option, usually to display or process additional details on the current object. This form will have user dialog (due to an **edit** or **display** statement in the <exec> trigger), and will usually require the identity (i.e.: primary key) of the current object to be passed as a parameter so that it can automatically retrieve that object.

A global proc called ACTIVATE_PROC (defined within SYSTEM_LIBRARY) has been created just for this purpose. It uses the following global variables:-

$$component	The name of the form component (default value is $fieldname).
$$instance	The name to be given to this instance, either the same as **$$component** (for a single instance) or a generated name (for multiple instances). Refer to section 5.7.6.4 *Choosing an Instance Name* for details
$$properties	To override the default settings within the component.
$$operation	Operation name within the component to be activated (the default is "EXEC" as only the <exec> trigger can contain an **edit** or **display** statement).
$$params	A single parameter string to be passed to the component (this is an associative list, therefore may contain any number of items).

Example: (<detail> trigger of a navigation button)

```
if ($empty(<MAIN>)
        message $text(M_90009)          ; nothing retrieved yet
        return(-1)
endif

$$component     = componentname         ; default is $fieldname
$$instance      = instancename          ; default is empty
$$properties    = propertylist          ; default is empty
$$operation     = operationname         ; default is empty
call LP_PRIMARY_KEY                     ; load values into $$params
call ACTIVATE_PROC                      ; activate component instance
```

Local proc module LP_PRIMARY_KEY is used to load the primary key of the current occurrence into global variable **$$params**, and contains code similar to the following:-

```
entry LP_PRIMARY_KEY    ; load parameter to pass to child process

$$params = $keyfields(<MANY>,1)         ; load names of pkey items
setocc "<MANY>",$curocc(<MANY>)         ; make occurrence current
putlistitems/id $$params                ; insert representations

end LP_PRIMARY_KEY
```

Note that in this example the operation being activated is the <exec> trigger, and the passing of parameters is one way. It is possible for this trigger to be activated many times with a different parameter string. This will cause the component structure of the child form to be cleared of its current contents before the new parameter string is processed.

It is possible to activate operations other than the <exec> trigger in a form component, and to have values passed back, but this will require proc coding similar to that which is described in the following section.

4.7.8.2 A Service component

The normal processing of a form may require the activation of an operation in a service component in order to perform a common function. A service does not have any dialog with the user (no **edit** or **display** statement is allowed) therefore an operation name other than "EXEC" may be used. As the operation may require more parameters than the single **$$params** string, the global proc NEW_INST_PROC should be used with the **activate** statement. This requires the following global variables:

$$component	The name of the form component.
$$instance	the name to be given to this instance, either the same as **$$component** (for a single instance) or a generated name (for multiple instances).
$$properties	To override the default settings within the component.

Example:

```
if ($formmodality(<instance>) < 0)          ; if instance not alive then instantiate it
        $$component     = componentname
        $$instance      = instancename
        $$properties    = propertylist
        call NEW_INST_PROC
        #include STD:FATAL_ERROR
endif

activate $$instance.operation(parameterlist…..)
#include STD:FATAL_ERROR
……
delete_instance $$instance
```

Note: For simple operations the call to NEW_INST_PROC may be omitted as the **activate** command performs an implicit **new_instance** if the component has not already been instantiated.

4.7.8.3 A Report component

No details yet.

4.7.8.4 Choosing an Instance Name

Instead of activating a component we create and activate an instance (copy) of that component. This allows more than one instance of the same component to be active at any one time. This is governed by the name that is chosen for each instance before it is created. If the instance name is set to the component name then only one copy of that component can exist. If multiple copies are required then different instance names need to be generated at run time.

Multiple instances are typical when creating child instances from a LIST form. The user selects the 1st occurrence and creates the 1st child instance, then selects a 2nd occurrence and creates a 2nd instance of the same child form. It is possible to get the system to generate a random and unique instance name, but this would allow multiple instances of the same child form being created for the same occurrence at the same time. It would therefore be advisable to construct each instance name using a value that links it with a particular occurrence. It would not be a good idea to use the occurrence number within the current hitlist as renumbering will occur if records are added or deleted. A value that is both unchangeable and unique for every database occurrence is the primary key, so it would seem logical to use this in the construction of the instance name. If the primary key value were to be used on its own this would prevent instances of different components with the same value from being active at the same time. A way to avoid this would be to use a combination of the component name and the primary key value.

A method has been designed into the menu system which allows the system administrator to choose whether to allow multiple instances of a component by defining the pattern to be used when constructing the instance name for that component. This is defined in the *instance_name* field for the component's entry in the MNU_TRAN table. A blank value (the default) signifies that the instance name should be the same as the component name. A non-blank value signifies the pattern to be used when constructing the instance name. The pattern should consist of the following:

- Any combination of letters, numbers or underscore characters.
- Field names enclosed in parentheses '(' and ')'. The field name must exist within the set of key values which is passed down from the parent form to the child.

UNIFACE insists that the instance name begins with a letter, and as some primary key values may be purely numeric it is advisable to specify an alpha prefix in the pattern. The instance name cannot exceed 16 characters, therefore rather than using the full component name as the prefix it would be advisable to use a shortened version. This name has three parts - <system prefix>, <sequence number>, and <dialog type>, therefore a unique prefix can be constructed from <dialog type> and <sequence number>.

For example, the menu system contains a transaction (component) called MNU_0020R which accesses a database entity whose primary key is USER_ID. The pattern for generating different instance names for each occurrence would therefore be "R20_(USER_ID)". At run time the pattern characters "(USER_ID)" will be replaced by the actual value from the database occurrence, so if the value was "FRED27" the generated name would be "R20_FRED27".

It is possible to specify more than one field name in the pattern. This is useful if the primary key is composed of more than one field.

Do not worry if the resulting instance name contains invalid characters or is more than 16 characters long – the standard software will strip out and truncate as necessary.

4.7.9 Invoking a 3GL routine

When invoking a 3GL routine perform the following:

- Initialise input and output parameters (subset of general-purpose variables $1-$99).
- Perform the 3GL routine, including resetting the input parameters.
- Check for any execution failure in **$procerror** – if this is non-zero then **$procerrorcontext** will contain full details of the error.
- Check for any non-zero value in **$status** (which can be set from within the 3GL routine).
- Use the output parameters.

Example: (using DDE to interface to an Excel spreadsheet)

```
$50 = "EXCEL"                    ; Excel service
$51 = "democar.xls"              ; .xls file containing spreadsheet
$52 = ""                         ; not used here
$53 = "r10c8"                    ; Cell: row 10, column 8
perform "uDdeRequest"            ; give me the data in that cell
if ($procerror)                  ; if (-1) then 3GL routine not found
        call PROC_ERROR($procerrorcontext)
        return(-1)               ; return corresponding value
else                             ; return value is zero
        total_amount.order = $53 ; contents of that cell
endif
```

4.8 COMMUNICATION BETWEEN OBJECTS

The activation of a modal form and the processing of return values can be done within the same context or proc. The same is true for activating a synchronous service. However, non-modal forms do not normally return values upon termination. Any values must be communicated by other means, as identified in the following paragraphs.

4.8.1 Via the ACTIVATE command

The ACTIVATE command will create the instance if it does not already exist. The instance must contain an operation name with a matching signature in order to receive any passed parameters.

4.8.1.1 Sending Parameters (to an ACTIVATEd module)

A component instance can pass parameters to another component instance by means of the following command:

```
activate instance.operation(parameterlist…..)
```

In order for this to be successful the receiving instance must have an operation defined with that name, and this operation must have a **params …. endparams** block that matches **parameterlist**. If an instance with the specified name does not currently exist an implicit **new_instance** statement is executed using the component name as the instance name. This instance will be created with default properties.

Although it is possible to have values passed back with this mechanism, if the instance is non-modal and is activated via its <exec> trigger which contains either an **edit** or **display** command the results will not be as expected. This is explained further in section *4.8.1.3 Using OUT or INOUT parameters in a child form*.

The standard processing for the <detail> trigger of navigation buttons assumes that the operation name is EXEC (for the <exec> trigger) and the parameter list contains a single string item that is input only. This is considered to be an associative list, and allows any number of values to be passed to the receiving instance. This is usually all the components of the primary key for an object which is to be retrieved from the database, but may include any other values that may be referenced within the receiving instance.

Where *parameterlist* contains a large number of entries it can be made more readable by adopting the following structure, which keeps each parameter on a separate line:

```
activate instance.operation   %\
      (parameter1             %\
      ,parameter2             %\
      ,parameter3             %\
      ,parameter4             %\
      ,parameter5             )
```

4.8.1.2 Receiving Parameters (in the ACTIVATEd module)

In order to receive a message the receiving instance must have an operation defined with the specified name, and this operation must have a **params …. endparams** block that matches every entry in **parameterlist**. For example:

```
<exec> trigger
params
        string      pi_param1       : IN
        string      pio_param2      : INOUT
        string      po_param3       : OUT
endparams
…….

exit(0)
```

The operation can be the <exec> trigger, or a named operation within the <operations> trigger. Note that operations **INIT**, **CLEANUP**, **ACCEPT** and **QUIT** do not have passed parameters.

Use one of the following prefixes on parameters so that they can be distinguished from local variables and entity items:

pi_	on IN parameters
po_	on OUT parameters
pio_	on INOUT parameters

4.8.1.3 Using OUT or INOUT parameters in a child form

When the **<exec>** trigger of a non-modal form is activated, and this trigger contains an **edit** or **display** statement, please keep the following in mind:-

- As soon as the **edit** or **display** is encountered the non-modal form will appear, but control will be passed back to the component instance that activated the non-modal form. The contents of **$status** will be zero at this point, signifying that the **activate** command was successful. All subsequent code in the trigger which issued the **activate** command will then be processed.

- Any statements in the **<exec>** trigger of the non-modal form that occur after the **edit** or **display** statement have **NOT** been executed at this point. This means that any attempts to assign values to OUT or INOUT parameters following the **edit** or **display** are not recognised.

- When control is returned to the activating component (following the **edit** or **display** statements in the non-modal form), the OUT and INOUT parameters have the values that they had when the **edit** or **display** statement was executed. This means that an OUT parameter can be 'undefined' if it was not assigned a value.

This problem can be avoided in the following way:-

- The parameters in the child form's <exec> trigger should be defined as IN only.

- When the child form is ready to pass the updated data back to its parent (eg: in the <accept> trigger) it should activate an operation in its parent, identifying the relevant objects in its list of parameters

- This operation in the parent form should define the parameters as IN only.

4.8.2 Via the POSTMESSAGE command

The POSTMESSAGE command can only be sent to an instance that already exists. The instance should contain the necessary code in its <async. interrupt> trigger in order to receive and process any message. If an instance of that name does not exist the message will be passed to the application <async. interrupt> trigger.

There is also a **sendmessage** command, but this cannot be used to communicate between instances that are contained within different applications being run on different paths, therefore our standard code uses the **postmessage** command only.

4.8.2.1 Sending a message

Within our standards when a child form is activated in order to modify the database (by adding, updating or deleting a database occurrence) it is usual to inform the parent process of any such update as soon as it has been committed.

Calls to the POSTMESSAGE proc are included within all those standard procs that commit changes to the database. This proc will be called only when the global variable **$$msgdata** contains a non-null value. The following variables are also used by the POSTMESSAGE proc:

$$msgdata	The message string. Where this identifies a change to the database it is usual to create this as an associative list containing all the fields from the changed occurrence, and not just the primary key.
$$msgid	An identifier for the message, which must not exceed 32 characters. If left blank this will be set to the contents of **$componentname**.
$$msgdst	Identifies the target instance for this message. If left blank this will be set to the contents of $**instanceparent**.

This is an example of the minimum code required to pass a message to the parent instance after a successful modification to the database.:

```
$$msgdata = ""            ; load message for parent
putlistitems/occ $$msgdata, "<MAIN>"
$$msgid     = “ENTRY_ADDED” | “ENTRY_UPDATED” | “ENTRY_DELETED”      ; choose one

call OK_PROC
```

To ensure that **$$msgdata** contains a complete list of all the fields from the occurrence, (ie: to avoid the results of subsetting) the field list for the entity should be set to ALL.

4.8.2.2 Receiving a message

The receiving instance (usually the parent) receives the message within its <async interrupt> trigger. The contents of global variables **$$msgid** and **$$msgdata** are received as functions **$msgid** and **$msgdata** respectively. The identity of the instance which generated the message is contained in function **$msgsrc**.

```
<async interrupt> trigger

if ($msgid = "ENTRY_ADDED")             ; entry added
        creocc "<MAIN>",$curocc(<MAIN>)+1
        getlistitems/occ $msgdata, "<MAIN>"
        call READ_INNER_ENT("<MAIN>")
endif

if ($msgid = "ENTRY_UPDATED")           ; entry updated
        creocc "<MAIN>",$curocc(<MAIN>)+1
        getlistitems/occ $msgdata, "<MAIN>"
        retrieve/o "<MAIN>"
        if ($status = 0)                ; does not exist
                discard "<MAIN>"
        else
                getlistitems/occ $msgdata, "<MAIN>"
                call READ_INNER_ENT("<MAIN>")
        endif
endif

if ($msgid = "ENTRY_DELETED")           ; entry deleted
        creocc "<MAIN>",$curocc(<MAIN>)+1
        getlistitems/occ $msgdata, "<MAIN>"
        retrieve/o "<MAIN>"
        discard "<MAIN>"
endif
```

It is assumed that **$msgdata** is an associative list constructed from all the fields of the affected occurrence. This will therefore contain not just the primary key but all the data.

Notice that there are no database retrievals to obtain the values for new or modified occurrences as this would cause any partially built hitlist to be completed. This could have a drastic effect on response times. The **retrieve/o** command is there simply to ensure that the occurrence identified by the primary key within **$msgdata** is made current.

The processing in this trigger will be performed for each message received, and is not restricted to a single message as the child process terminates.

As the results of additions and modifications are not retrieved from the database but overlaid within the component's external structure, the default processing within the <leave modified key> and <validate key> triggers would reject these occurrences for being duplicates. For this reason these triggers should be disabled.

4.8.3 Via the CALL command

4.8.3.1 Sending Parameters (on the CALL command)

Values to be passed to and/or passed back from a proc module can best be specified as a list of arguments on the command line, as in the following example:

```
call EXAMINE_REPLACE(string, valueA, valueB)
if ($procerror)
        call PROC_ERROR($procerrorcontext)
        return(-1)
endif
if ($status)
        ….              ; conditional processing
endif
; STRING now has all occurrences of VALUEA replaced with VALUEB
```

In order for this to be successful a global or local proc module with that name must exist, and this module must have a **params …. endparams** block that matches the argument list. If there is any failure the call to PROC_ERROR will append the details to the message frame. Note that the proc module may return a value in **$status** to indicate the results of its processing.

Upon completion of the called module some of these parameters may have different values. This can only be determined by the *direction* indicator against each of those parameters within the called module.

Where *parameterlist* contains a large number of entries it can be made more readable by adopting the following structure, which keeps each parameter on a separate line:

```
call proc_module        %\
        (parameter1     %\
        ,parameter2     %\
        ,parameter3     %\
        ,parameter4     %\
        ,parameter5     )
```

4.8.3.2 Receiving Parameters (in the CALLed Procedure)

Values to be received by the proc module are specified in a **params …. endparams** block. Each entry has a *direction* indicator which can be one of IN, INOUT, or OUT.

```
entry EXAMINE_REPLACE          ; examine STRING replacing OLD with NEW
params
        string      pio_string_in       : INOUT
        string      pi_text_old         : IN
        string      pi_text_new         : IN
endparams
….
….
return(0)
end
```

Parameters marked as IN or INOUT should have values supplied by the calling object.

Parameters marked as OUT or INOUT may have their values updated before control is returned to the calling object.

Use one of the following prefixes on parameters so that they can be distinguished from local variables and entity items:

pi_	on IN parameters
po_	on OUT parameters
pio_	on INOUT parameters

4.9 DATA VALIDATION

4.9.1 New triggers

Data validation within components has been enhanced with the following triggers that became available with UNIFACE version 7.2.01:

<validate field>	After declarative checks, and before the <leave field> trigger is activated.
<validate key>	Before the <leave modified key> trigger is activated.
<validate occurrence>	After declarative checks, and before the <leave modified occurrence> trigger is activated.

If the code within these triggers sets **$status** to a negative value then the associated <on error> trigger will be fired.

Entities which were created in earlier versions will not contain any code within the <validate key> trigger. To take advantage of the additional processing the application model should be modified to include a call to global proc VLDK_TRIGGER. Automatic validation of candidate keys will not be performed unless the Validate property for that key is clicked *ON* in the Define Key form.

Standard validation rules can be defined in the <validate field> and <validate occurrence> triggers within the application model, and will therefore be inherited by all components which reference them. These triggers will be automatically invoked whenever the associated object is modified, or when specifically requested in proc code. Provided that component-specific validation is defined within the existing <leave field> and <leave modified occurrence> triggers inheritance from these application model defaults will not be lost.

4.9.2 $dataerrorcontext

The details of any data validation errors are now recorded in function **$dataerrorcontext**. A new global proc has been provided (DATA_ERROR) that will append these details to the message frame to aid in any debugging efforts. This proc will be called automatically by the existing ON_ERROR_E (entity) and ON_ERROR_F (field) procs, therefore additional programming effort is not required.

Here is an example of the output from proc DATA_ERROR:

```
CONTEXT=VALIDATION
TOPIC=KEY
OBJECT=MNU_TRAN.MENU
OCC=1
KEY=1
ERROR=147
STATUS=-1
DESCRIPTION=0147 - Validation failed for key.
```

4.10 VERSION TRACKING

4.10.1 Version History

Various objects should contain a block of text similar to the following to aid in documentation.

- For components this should be in the Comments field of the component properties.
- For global procs this should be at the end after all the code.

```
;------------------------------------------------------------------------
; Name:                 HELPA_PROC
;
; Description:
;
; Author:               Tony Marston
;
; Date written:         01-07-99
;
; Current Version:      1.0.0
;
; Input parameters:     $formname
;                       $fieldname
;
; Output parameters:    none
;
; Update History:
; Date          Updated By      Details
; 02-07-99      Tony Marston    Replaced RUN with ACTIVATE.
```

Whenever the object is amended the following should occur:-

- The object's version number should be incremented
- A brief description of the change should be added to the history at the bottom.

4.10.2 Version Numbers Within Forms

During the life of a software module it is more than likely that it will be subject to numerous updates, whether these are to fix bugs or to incorporate changes or enhancements requested by the user. As it is more than likely that several copies of the software will exist in numerous places (eg: the development environment, the test environment, the delivery environment, and one or more user sites) it becomes necessary to have a mechanism whereby it is easy to check that a module being run is the latest version available.

To this end every form will have its own version number - this will be held in a local variable (name = **$form_version$,** type = string) which must be set to the current value as part of the initial processing. The default INIT operation obtains this value from a local constant, therefore there is no need to change the operation and loose inheritance from the component template.

This version number will be in the form **AA.bbb.ccc** where

AA	is the major version number, starting at 01 and incremented by 1 for each major change requested by the user.
bbb	is the minor version number, starting at 000 and incremented by 1 for each minor change requested by the user. This is reset to 000 whenever the **AA** portion is incremented.
ccc	is the count of bug fixes, starting at 000 and incremented by 1 for each bug fixed. This is reset to 000 whenever the **AA** portion is incremented.

When a module is first released its version number should be 01.000.000.

This version number **must** be updated each time the module is changed, and precise details of the change **must** be documented in the version history that can be found in the COMMENTS field of the form's properties.

The version number can be interrogated at run time by selecting the HELP -> HELP ABOUT option from the pulldown menu bar.

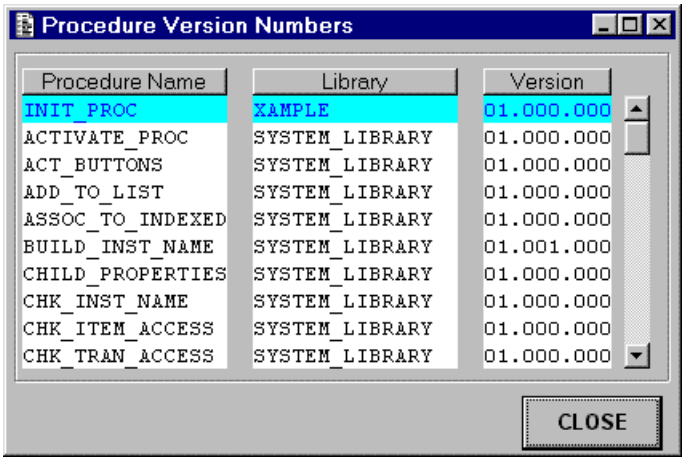
4.10.3 Version Numbers Within Global Procs

In order to provide a mechanism that will display the current version numbers of the global procedures the following code should be inserted at the start of every procedure:-

```
if ($$version_only)                    ; is flag set ?
        $1 = "01.000.000"              ; version number
        $2 = "SYSTEM_LIBRARY"          ; library
        return(0)
endif
```

The format of the version number is as described earlier. This **must** be changed each time the procedure is modified.

See template form X_VERS for an example of how the procedure version numbers can be extracted and displayed. This will produce a screen similar to the following:-



4.11 USE OF SELECTDB

The SELECTDB statement is used to provide a single value (or set of values) from any number of occurrences within the database. Here are some typical examples:-

A) Find the highest value used for CUSTOMER_NO to reset a counter:

```
$1 = 0
selectdb max(customer_no) from "customer"                    %\
         u_where (id_sman.customer = id_sman.salesman)       %\
         to $1
if (last_cust_no.salesman != $1) last_cust_no.salesman = $1
```

B) Accumulate values from each line in an invoice to give a total value:

```
$total$ = 0
selectdb sum(total_price) from "invoice_detail"              %\
         u_where (invoice_id.invoice_detail = invoice_id.invoice)  %\
         to $total$
```

For functions such as COUNT, SUM, MINIMUM, MAXIMUM and AVERAGE it is more efficient to use the SELECTDB command than to retrieve all the occurrences into the form (or a separate hidden form). In this way all the hard work is performed by the database, with only the result(s) being passed back to the form. If multiple occurrences are passed back to the form for processing then this increases the amount of network traffic, which could contribute to poor response times.

1. Always initialise any output fields before invoking this command as the previous contents may not be overwritten if zero occurrences match the selection criteria.

2. If the **u_where** clause does not include **column_name = value** where *column_name* is an indexed field and *value* is other than null, this will initiate a full table scan. Range checks (less than, greater than) on indexed fields may cause the index to be scanned rather than the table itself, but this depends on the underlying database.

3. Be aware that the use of a **u_where** clause which includes **column_name = “”** may cause the underlying DBMS to ignore any indexes and initiate a full table scan - on tables with many entries this has a serious effect on performance.

4. The SELECTDB statement cannot be used in any components where the entities are accessed by Object Services.

4.12 ACTIVE OBJECT HIGHLIGHTING

4.12.1 Active Field

The field that has focus can be highlighted by defining a value for the **$active_field** setting in the .asn file. This relies on standard UNIFACE processing to automatically set the current field to the specified colour, and to reset it as soon as focus changes to a different field.

This applies to every field within the system, except for those which have the **cursor video** option in the layout properties turned off. This typically applies to commandbuttons and radiogroups.

4.12.2 Active Occurrence

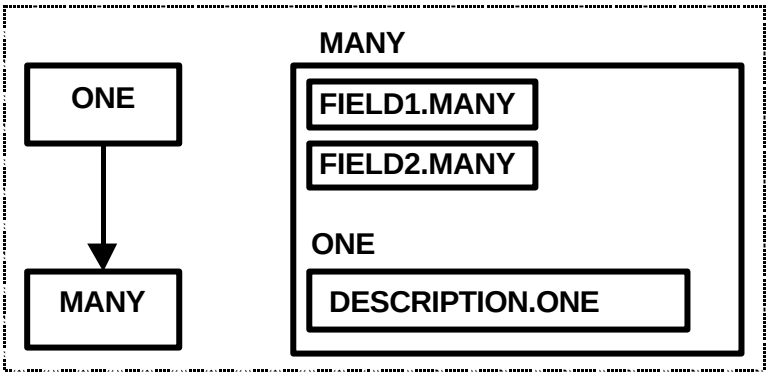
In a form component that contains multiple occurrences the one that currently has focus can be highlighted by defining a value for the **$def_curocc_video** setting in the .asn file. This relies on standard UNIFACE processing to automatically set all fields of the current occurrence to the specified colour, and to reset them as soon as focus changes to a different occurrence.

The **$curocc_video** setting in the .asn file would automatically apply this colour to all occurrences within the application, however this has not been found to be 100% reliable. Instead use is made of the **curocc_video** command within the <exec> trigger.

4.13 POPUP PROCESSING

4.13.1 Overview

This form of pick-list processing is required when two entities are defined in a ONE-to-MANY relationship, but are painted on a form with MANY as the outermost entity and ONE as an inner entity (thereby forming a ONE-to-ONE relationship).



The ONE entity contains two fields: **primary_key** and **description**. It is possible for it to contain more fields, but only these two are relevant in this example.

The MANY entity has its own primary key, and among its other fields is one called **foreign_key,** which is used to relate it to the ONE entity. In these circumstances the entity ONE is known as the foreign entity.

The screen does not contain either of the fields **primary_key.one** or **foreign_key.many**, but does contain **description.one**. The description is often more meaningful to the user than an obscure code.

Because the field **foreign_key.many** it not displayed on the screen its value cannot be changed directly by the user, it can only be changed by invoking a popup form. This is a special form, usually read-only, which lists the contents of the ONE entity and allows the user to choose one of the occurrences. This causes the primary key of the chosen occurrence to be passed back to the calling form, which is then able to **retrieve** it, filling in the **description** field on the screen, and transporting the primary key value down to **foreign_key.many**.

4.13.2 POPUP Invocation

A popup form can be invoked in any of the following ways:-

a) By "double-clicking" on an empty DESCRIPTION field. (NOTE: "double-clicking" fires the <detail> trigger, which is the same as using the fast key “<control>d”)

b) By "single-clicking" on the POPUP icon (containing an "up" arrow).

c) By entering any profile characters in the DESCRIPTION field, and by "double-clicking" on either the field itself, or the icon.

If the user modifies the contents of the DESCRIPTION field and leaves the field without invoking the popup in one of the prescribed manners, the field will be cleared and the associated foreign key value will be set to null.

The profile (the contents of the DESCRIPTION field) is always passed to the popup form, even if it is empty. This is a single string parameter, but as it is an associative list it can contain references to several fields.

This profile is used by the popup form to retrieve entries from the relevant database tables.

If no entries are found the popup screen is not shown, and an error message is issued. If only one entry is found its primary key is automatically passed back without any need to show the popup screen. If more than one entry is found the popup screen is shown so that the user can pick one of the entries. The user can leave a popup screen in one of the following ways:-

a) By putting the cursor on the required occurrence and pressing the 

button. The popup form will exit and pass back details of the selected entry to the calling form, which will retrieve that entry and display the relevant details. The cursor will then move on to the next field.

b) By double-clicking on the required occurrence (invoking the <detail> trigger). This has the same effect as pressing the 

button.

c) By pressing the CANCEL button. The popup form will exit, no details will be passed back to the calling form, a warning message will be issued, and the prompt will be positioned on the DESCRIPTION field.

If an occurrence is selected within a popup form its primary key is loaded into global variable **$$selection** as a list of field names and values. This automatically caters for primary keys that consist of multiple items.

4.13.3 POPUP Coding

a) Form Contents

Define entity ONE inside the boundaries of entity MANY.
Define field **description.one** to the required size.
Ensure the field properties of **description.one** include "double click = <detail>".
Define a pushbutton immediately to the right of **description.one** (height = 1, width = 2), and rename it as POPUP_BUTTON. If this has not been defined in the application model then apply field template POPUP_BUTTON to establish the default settings.

b) <ON ERROR> trigger for FOREIGN_KEY.MANY

```
$prompt = "description.one"                    ;***** change this line
call ON_ERROR_F
```

Note that the fieldname on the **$prompt** statement is enclosed in quotes - if it is not this will generate a compiler warning (or an error under version 6.1d) on those forms which do not actually contain the named field.

If the **$prompt** statement is inserted into the application model this will avoid the necessity of having to paint it on the form so that the trigger can be amended locally. Due to a feature in UNIFACE this would also require you to paint **primary_key.one** on the form otherwise no value will be transported to **foreign_key.many** - this will also produce a compiler warning saying that a key field has been painted more than once.

c) <START MOD> trigger for DESCRIPTION.ONE

```
release/e
```

d) <DETAIL> trigger for DESCRIPTION.ONE

```
if (fieldvalue != "")                          ;***** optional
        putitem/id $$profile, "fieldname", fieldvalue   ;***** optional
endif                                          ;***** optional
call POPUP_DTL("popup_form")                   ; change "popup_form"
#include STD:FATAL_ERROR
if ($status = 0)
        macro "^NEXT_FIELD"
endif
```

By default the profile passed to the popup form will contain the name and value of the current field only. If additional values are required then insert as many of the optional **putitem** statements as necessary in front of the call to POPUP_DTL.

e) <LEAVE FIELD> trigger for DESCRIPTION.ONE

```
call POPUP_LFLD
if ($status) return(-1)
```

f) Definition of POPUP Button

The POPUP pushbutton can be included in the field list within the Application Model for each relevant entity - as it is a non-database field it will not be included in any data transfers. It should be painted on the form immediately after the **description.one** field, and should contain the attributes listed below. These should not need amending:-

Description	popup button
Initial value	^U_POPUP_BUTTON (identifies a Glyph entry)
Data type	Image
Widget type	Commandbutton
Interface template	POPUP_BUTTON
Syntax template	POPUP_BUTTON
Layout template	POPUP_BUTTON
Characteristics	Boilerplate
<detail> trigger	call POPUP_BTN_DTL

4.14 LOGICAL UPDATES ACROSS MULTIPLE FORMS

There may be occasions within a system where a logical transaction (a series of updates that are to be considered as a single unit) needs to be spread across multiple forms. The approach taken depends on how many of the forms in the group require dialog with the user.

4.14.1 Multiple forms with Single Dialog

This is where a form with user dialog does some updating of its own, but is required to call one or more hidden forms (ie: without any user dialog) to carry out some additional updates. Any one of these subordinate forms may also call other hidden forms, thus producing a hierarchy that may be several levels deep.

To ensure that database integrity is maintained throughout this partitioned update, and that any impact on database performance is kept to a minimum, the following points should be observed:-

a) Once the update sequence has started it must not be interrupted by any dialog with the user. This means that only the primary (parent) form can contain an EDIT statement - all of the subordinate (child) forms should be of type HIDDEN (ie: no dialog with the user).

b) The parent form must STORE its own changes without a COMMIT before calling any subordinate forms.

c) No subordinate form should perform either a COMMIT or a ROLLBACK, as these must be performed by the parent form as and when necessary.

d) There must be only a single COMMIT to cover the entire update sequence, and this must be performed by the parent form only after successfully returning from all subordinate forms.

e) There must only be a single ROLLBACK to cover the entire update sequence in the event of an error, and this must be performed by the parent form. It follows therefore that if any failure is detected in any subordinate form then control must be returned immediately to the parent form with an appropriate value set in **$status**.

The existing procs (STORE_PROC or OK_PROC) cannot be used in these circumstances as they contain an automatic COMMIT after a successful STORE. This has necessitated the creation of the following additional central procs:-

1) **STORE_NO_COMMIT**

   This performs a STORE (without a COMMIT or ROLLBACK), and returns a status code of either 0 (OK) or -1 (failed). It also does not display any message in the message window.

2) COMMIT_PROC

   This performs a COMMIT and returns a status code of either 0 (OK) or -1 (failed). A corresponding message is displayed in the message window.

3) ROLLBACK_PROC

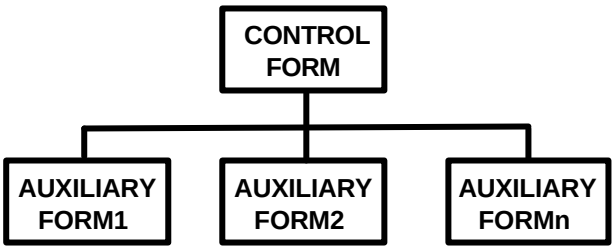
   This performs a ROLLBACK and issues the STORE FAILED message.

4) Sample <ACCEPT> trigger for the Parent form

```
call STORE_NO_COMMIT

if ($status = 0)
    activate "FORMB"                ; additional updates (1)
endif

if ($status = 0)
    activate "FORMC"                ; additional updates (2)
endif

if ($status = 0)
    call COMMIT_PROC                ; commit all changes
endif

if ($status = 0)
    exit(0)                         ; AOK
else
    call ROLLBACK_PROC              ; undo all changes
    return(-1)
endif
```

4.14.2 Multiple Forms with Multiple Dialog

This is where a logical transaction is split across a group of forms, each having dialog with the user. This may be used, for example, where an entity contains so many fields that they cannot all be processed on a single form. In this case the group will be comprised of a single Control form and one or more Auxiliary (or Overflow) forms. This can be represented in the following structure:-



This structure has the following characteristics:-

a) All database activity (read, write, store, commit, and rollback) is handled by the Control form. All entities and occurrences to be included in the logical transaction must be defined within the Control form. This means that the decision to accept or cancel the update(s) is handled within the Control form alone.

b) Auxiliary forms do not access the database (except for lookups on foreign entities) therefore all relevant triggers can be disabled. The form type should be set to LIMITED.

c) Control forms may deal with multiple occurrences of multiple entities, but Auxiliary forms may only deal with a single occurrence of a single entity (except for lookups on foreign entities).

d) Communication between the Control form and an Auxiliary form is via a single string variable which is an associative list. The data can be inserted with the **putlistitems/occ** command, and retrieved with the **getlistitems/occ** command.

e) Auxiliary forms may be called in any sequence, and as many times as required.

f) Auxiliary forms can only be called from Control forms - they cannot be called directly from a menu.

The Control form requires code which achieves the following:-
- Activates a child form and passes the current data values to it.
- Has a method of receiving updated data as and when required (one of the following):
  - As an operation which can be activated by the child, or
  - By receiving a message in the <async interrupt> trigger via the **postmessage** command.

The auxiliary form requires code which achieves the following:-
- Receives data when activated by the parent.
- Displays the data and allows the user to make changes.
- Passes back the data to the parent only if the OK button is pressed (if CANCEL is selected then no data is returned).

4.15 ONLINE HELP

Every online form should have a pulldown menu bar that contains a HELP option. When selected this should offer the following options:-

- Show Help
- Keyboard Map
- About...

4.15.1 Show Help

The <option> trigger contains the code **macro "^HELP"**, which fires the <help> trigger of the current field or entity. Each of these triggers contains a call to the standard HELP_PROC, which contains the following code:-

```
putitem/id parameters,"formname",$componentname
putitem/id parameters,"entname",$entname
putitem/id parameters,"fieldname",$fieldname
activate "%%$variation%%%_HELP".EXEC(parameters)
```

An example form is provided with the name XAMPLE_HELP.

This will fill the display area with help text for the field identified in **fieldname**. Pushbuttons are available to ZOOM into this field, or to alternate the display between text for **formname** and text for **fieldname**.

The creation of help text is normally the responsibility of the user as text created by technicians (the development team) may not be readily understood by non-technicians (the users).

It is now possible for online help to be provided from external HTML files – please refer to to HELP_PROC in Appendix G for more details.

4.15.2 Keyboard Map

The <option> trigger contains the code **macro "^KEY_HELP"**, which is a standard UNIFACE function to display the current keyboard map. No further action is required.

4.15.3 About...

The <option> trigger contains the code **call HELP_ABOUT**, which contains the following code:-

```
putitem/id parameters,"form_name",$componentname
putitem/id parameters,"form_title",$formtitle
putitem/id parameters,"form_version"$form_version$
activate "%%$variation%%%_HELPA".EXEC(parameters)
```

An example form is provided with the name XAMPLE_HELPA.

The system version number and release date will need to be set manually. They should be updated each time a set of modules is released, either to system testing or to the client.

4.16 OBTAINING NEXT NUMBER IN A SEQUENCE

There is sometimes the need to obtain the next number in a sequence, usually to provide a value for a primary key. There are various different methods available, as described below.

4.16.1 Uniface Counters

UNIFACE provides the ability to maintain sets of counters inside the UOBJ file. These can be created by the **numset** command and incremented by the **numgen** command, as shown in the following sample procedure:-

```
entry GET_NEXT_NUMBER
params
        string      PI_COUNTER_NAME      : IN
        numeric     PO_COUNTER_VALUE     : OUT
endparams

numgen PI_COUNTER_NAME, 1, $variation          ; increment counter
if ($status < 0)                               ; name not found
        numset PI_COUNTER_NAME, 1, $variation  ; create counter, start at 1
        if ($procerror)
                call PROC_ERROR($procerrorcontext)
                rollback "$UUU"
                return(-1)
        endif
endif

commit "$UUU"                       ; update UOBJ

PO_COUNTER_VALUE = $result  ; return result

return(0)

end GET_NEXT_NUMBER
```

This procedure can be called using code similar to the following:-

```
call GET_NEXT_NUMBER("INVOICE_ID", invoice_id.invoice)
if ($procerror)
        call PROC_ERROR($procerrorcontext)
        return(-1)
endif
```

However, using UNIFACE counters does have the following drawbacks:-

a) The update of UOBJ is committed before any updates to the application database, therefore if there is any failure with the application updates the counter cannot be rolled back to its previous value, especially if other users have incremented the same counter in the mean time.

b) Each time a counter is incremented the UOBJ file must be locked - this may cause problems if there are a large number of users all trying to increment counters at the same time.

c) If the UOBJ file is ever replaced (eg: following the release of a new version of the application) all the counters are lost. These may be re-instated manually, but this presupposes that the latest counter values were obtained before the UOBJ file was replaced. It may be wise to create a

special form that uses SELECTDB statements to identify the highest values found on the application database and to reset the counters accordingly.

d) Users of the same application database must access the same set of counters in the same UOBJ file, otherwise the same value will eventually be obtained from different sequences, resulting in a *duplicate primary key* error when trying to update the database. If, for example, groups of users in a large network are accessing different (local) copies of UOBJ then UNIFACE counters cannot be used without the possibility of producing a duplicate number.

e) Users who share the same UOBJ file must also share the same application database. In the situation where there are different copies of the application database (eg: for development, system testing, user training, live) there will be gaps in the numbers used on any one copy of the application database. This may cause confusion to the users, or may cause the field size to be overflowed sooner than expected.

Due to these problems it is strongly recommended that UNIFACE counters **not** be used.

4.16.2 Runtime Retrieval

This method does not use a separate field on the database to hold the counter value. Instead it uses the **selectdb** statement at the appropriate time to identify the highest number used so far on the application database so that it can be incremented, as in the following example:-

```
$1 = 0
selectdb    max(customer_no) from "customer"                    %\
            u_where (sman_id.customer = sman_id.salesman)       %\
            to $1
customer_no.new_customer = $1+1
```

This method does, however, have the following drawbacks:-

a) Where the field is not indexed multiple database records will have to be scanned in order to identify the highest number used, and the time taken to complete this statement will increase in direct proportion to the number of entries on the specified database table.

b) If the form is capable of creating more than one new occurrence at a time then this increases the areas that need to be searched in order to identify which is the highest value used so far - the database and the form's own external structure. This increases the complexity of the code, and therefore increases the possibility of error.

c) It is possible that in the interval between executing the **selectdb** and actually performing the **store** another user has crept in and used the same number. In order to avoid the update failing with a *duplicate primary key* error the following code could be inserted into the <write> trigger:

```
write
while ($status = -7)                        ; duplicate primary key
      customer_no = customer_no + 1         ; try next number
      write
endwhile
```

If the field name used is indexed then the DBMS may scan the index rather than the table, which will reduce the response time. In the example where the primary key of CUST_ADDRESS is a combination of CUST_ID and ADDRESS_NO, the following code can be used to provide the next available value for ADDRESS_NO:

```
$1 = 0
selectdb max(address_no) from "cust_address" %\
      u_where (cust_id.cust_address = cust_id.customer) to $1
address_no.cust_address    = $1 + 1
```

4.16.3 Database Triggers

The underlying DBMS may support a function within a trigger that can generate the next available value for a primary key field. This may be an efficient method, but it does present the following disadvantages:-

1. Not all DBMS's support this functionality, therefore the choice of database engines for this application would be limited.

2. The value supplied by the DBMS would not be passed back to the UNIFACE form as part of the **store** operation, therefore this mechanism cannot be used if the value is to be used for another entity's foreign key within the same operation.

3. As database triggers are defined manually within the database and are outside the control of UNIFACE this means that application logic is split across separate environments, which could lead to possible documentation and maintenance difficulties.

4.16.4 Database Procedures

The underlying DBMS may support a function within a procedure that can generate the next available value for a primary key field. Unlike a trigger a procedure is called from within a UNIFACE component and returns its result back to that component. However, it does present the following disadvantages:-

1. Not all DBMS's support this functionality, therefore the choice of database engines for this application would be limited.

2. As database procedures are defined manually within the database and are outside the control of UNIFACE this means that application logic is split across separate environments, which could lead to possible documentation and maintenance difficulties.

4.16.5 Database Counters

This method uses counters that are located within the application database itself, thus avoiding some of the problems with UNIFACE counters in a separate UOBJ file. These counters can be located in any of the following places:-

4.16.5.1 Single Control Record

The control table contains a single record, but this contains a separate field for each counter. Every time a new number is required from any counter the single control record must be locked, which will cause delays and conflicts in a system with a large number of users. This method should therefore be avoided.

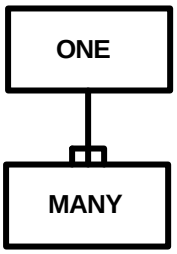
4.16.5.2 Multiple Records in a Single Control File

There is still a single control file, but this time it contains a separate record (occurrence) for each counter. Each record contains an identity (counter name) and value (last number used). This method causes fewer locking conflicts as it is only when the same counter is updated that the same record is locked.

4.16.5.3 Multiple Records in Multiple Files

When a counter value is used as a single-item primary key because that number must be unique across the whole database (eg: a customer number, an invoice number) then that counter must be obtained from a single source.

When a counter value is combined with another field (or fields) to produce a compound key the location of this counter should be the record where the other field (or fields) form the primary key.



Contains the field LAST_SEQ_NO

Related to ONE entity via field FOREIGN_KEY
Primary Key = FOREIGN_KEY + SEQ_NO

In this example the primary key of the MANY entity is comprised of the foreign key field (which relates it to the ONE entity) and a field called SEQ_NO. New values for SEQ_NO (MANY) are obtained by incrementing the current value in LAST_SEQ_NO (ONE). Note the use of LAST instead of NEXT - when an occurrence of ONE is created the initial value of LAST_SEQ_NO is zero (or null). The code to obtain values for SEQ_NO should be similar to the following:-

```
<accept> trigger

retrieve/e “one”                                  ; ** only if not already retrieved **
last_seq_no.one  = (last_seq_no.one + 1)          ; increment
seq_no.many      = last_seq_no.one                ; use

call OK_PROC                                      ; update both MANY and ONE entities
```

This processing should take place at the last possible moment (ie: immediately prior to the **store** command), therefore it may be necessary to assign a temporary value to SEQ_NO to avoid the problem caused by leaving a required field with a null value. This may be done with code similar to the following:-

```
<occurrence gets focus> trigger

if (seq_no.many = “”)
        seq_no.many/init = 0            ; assign dummy value
endif
```

This has the advantage that it is only when creating occurrences of the MANY entity for the same ONE entity that there will be any locking conflict.

Another bonus is that if it is required to retrieve only the latest occurrence of the MANY entity, this can be done in a single operation as all the components of the primary key are available.

It may be possible to locate the ONE entity in the form where it is not related to the MANY entity (eg: at the bottom), in which case the occurrence need not be retrieved until the update is confirmed.

4.17 LIST PROCESSING

This is where the user must choose a value from a pre-defined list rather than inputting free-format text. At present a list can be retrieved from the following sources:-

⇒ From the Application Database.
⇒ From the Application Model.
⇒ From the Application Message File.

4.17.1 Application Database

In this option each list is defined as a separate entity within the application model. The entity normally contains only a brief code and a meaningful description. All references to this list will be identified by a relationship within the application model. This option also requires a user function to maintain the list of values as well as a read-only function (a popup form) which displays the list of available values and allows the user to choose one.

This option is normally reserved for the following circumstances:-

⇒ Where there are a large number of values.
⇒ Where the number of values can be readily changed by the user.
⇒ Where the values do not have any affect on any subsequent processing.

The advantages of this option are:-

⇒ The contents of the list can be altered at any time by the user using the maintenance function provided.

The disadvantages of this option are:-

⇒ If the software requires certain values an installation procedure is required when creating a new (empty) copy of the application database.

⇒ Should not be used in those circumstances where specific values are set by or tested for within the software, as the user could change the contents of the list which, without the corresponding changes to the software, could lead to confusion.

⇒ It is not possible to have foreign language equivalents of the same code on the same application database (*unless* language code is included as a data item).

Replacing Popup form with DropDownList processing

Instead of having to create a popup form to cater for the picklist processing it is possible to utilise the functionality of dropdownlists (or even radiogroups) on specific forms. However, this does require some additional processing to transfer the contents of the list from the database into the widget properties of the field at run time. This can be achieved with the following steps:-

a) Create a hidden form to transfer the database contents into a global variable (a separate variable for each list). This form need contain only the database entity in question, and the <execute> trigger should contain code similar to the following:-

```
$list$ = ""                                    ; initialise list

retrieve

setocc "ENTITY",1

putlistitems/id $list$, id, description ; add entry to list from all occurrences

$$v_ENTITY = $list$                            ; load into global variable

exit(0)
```

b) This hidden form must be run at least once in each session in order load the values into the global variable. This may be done within the <execute> trigger of each form that references the list using code similar to the following

```
if ($$v_ENTITY = “”)                    ; list is empty
        activate “hidden form”          ; extract from database
endif
```

Alternatively it may be done en masse as part of the application initialisation procedure (see global procedure INIT_PROC).

```
if (!$$first_time_flag)                 ; has this been run yet?
        $$first_time_flag = 1           ; no, but it has now
        activate “hidden form”          ; extract from database
endif
```

c) The contents of the global variable can be loaded into the relevant field at run time using code as described in section *4.17.4 Manipulation of List Contents at run time*.

This option Is not advisable where the contents of the table are subject to frequent changes as the global variable can only contain the values that were available when the hidden form was run. Any changes made to the database table after this time will **not** be included in the variable until the hidden form is re-run. Unless a mechanism is incorporated into the system to initialise the variable or re-run the hidden form the only option would be to terminate the session and restart it..

4.17.2 Application Model

In this option any reference to the list is not defined as a foreign key linked to the list entity, but as a field with a widget type of RadioButton or DropDownList. The values for the list are hard-coded within the application model, and do not require any extra coding when included in any form.

This option is normally reserved for the following circumstances:-

⇒ Where there are only a small number of values.
⇒ Where the range of values is fairly static (ie: not changed very often, if at all).
⇒ Where particular values are set or tested for by the application software.

The advantages of this option are:-

⇒ No installation procedure is required on a new (empty) application database.

⇒ No additional forms are required to maintain or examine the list.

⇒ Changes can only be made by the system developers, so can be synchronised with corresponding changes to the software.

The disadvantages of this option are:-

⇒ It is not possible for the user to make any changes to the list.

⇒ If any changes are made to the list then all forms which reference the list must be re-compiled, otherwise they will continue to display the previous list.

⇒ It is not possible to have foreign language equivalents of the same list within the application model.

4.17.3 Application Message File

In this option the list is defined as a RadioButton or DropDownList, but the values are obtained from the message file at run time.

The disadvantages of this option are:-

⇒ Can only be used where the contents of the list are static and pre-defined (eg: status values).

⇒ Can only be used where the list contains two values - an identity and a description.

⇒ Changes to the list which affect the number of items must be synchronised with corresponding changes to the software.

The advantages of this option are:-

⇒ Minor changes (eg: to descriptions) can be made to the list without having to re-compile any forms.

⇒ It is impossible for the user to change, and possibly corrupt, the contents of the list.

⇒ As lists of values are maintained within the software, not the application database, then no installation procedure is required whenever a new copy of the application database is created.

⇒ The message file can contain copies of the list in different languages, with the version being retrieved at run time being determined by the contents of **$language**.

This option does, however, require some additional procedure code, but this can be defined centrally within the application model rather than being dealt with separately within each form.

In the following example the entity AUTHORITY has a field called STATUS which can have one of the following values - 1=Outstanding, 2=Accepted, 3=Rejected. The steps to implement this are as follows:-

a) Define the field within the application model

The field must be defined with an interface definition suitable for the code values, not the descriptions. The widget type need not be set at this time as it can be defined as required on the relevant form. It is possible for the same field to be defined as an Editbox on one form, a RadioButton on another, and a DropDownList on a third.

b) Define an entry in the message file

The name of the entry should contain the prefix of “LIST_”. The entire list of values must then be defined in the message text with each value being separated by <GOLD>semicolon, for example

```
1=Outstanding+;2=Accepted+;3=Rejected
```

c) Transfer to a Global Variable

This avoids the need to access the message file more than once for the data. This requires code to be inserted into the <exec> trigger of each relevant form, or after the test for **$$first_time_flag** within global procedure INIT_PROC, similar to the following:-

```
$$v_auth_status = $text(list_auth_status)                    ; get list
```

The contents of the global variable can be loaded into the relevant field at run time using code as described in the following section.

4.17.4 Manipulating List Contents at run time

The list of internal values and their associated external representations should be held in a global variable. The contents of this variable can be obtained either from the application database, or from the message file, as demonstrated in the previous sections.

a) Load complete list into field's widget properties

```
$valrep(field.entity) = "%%$$global_variable"          ; all occurrences
        or
$fieldvalrep(field.entity) = "%%$$global_variable"     ; current occurrence
```

If the form contains multiple occurrences of the named entity a single call to $VALREP will load the list into ALL occurrences. A call to $FIELDVALREP will only load the list into the current occurrence.

b) Making a dropdownlist display only

Changing the field syntax of a dropdownlist to "NED" will prevent the user from changing the current value, however it does not prevent the user from scanning the list to see what other alternatives are available. It is only when attempting to select a different option is the user informed that it cannot be changed. To avoid this problem a custom widget has been created called fDropNoEdit which has the following properties:-

```
fDropNoEdit=udropdownlist(3d=off;entries=0;forcefit=on;dynamic=on;autoselect=off)
```

The "entries=0" part allows the current value to be displayed, but disables the remainder of the list.

d) Temporarily remove an item from the list

There may be circumstances in which a particular value should not be available for selection, in which case it can be removed from the current occurrence (without affecting the contents of the global variable) with the following code:

```
delitem/id $fieldvalrep(status.authority), "value"
```

e) Temporarily add an item to the list

There may be circumstances in which an additional value should be available for selection, in which case it can be added to the current occurrence (without affecting the contents of the global variable) with the following code:

```
putitem/id $fieldvalrep(status.authority), "value", "representation"
```

4.18 HITLIST PROCESSING

If a form has the potential for dealing with stepped hitlists, and this form can activate child forms that retrieve occurrences from the same database entity, this can lead to potential performance problems. There are various options for dealing with the hitlist before control is passed to the child process. These are configurable on a transaction-by-transaction basis – the chosen option can be specified in the CHILD_PROPERTIES field for the parent form on the Menu database

The available options are as follows:

<blank>	Default UNIFACE behaviour. If the hitlist in the parent process is incomplete (ie: there are entries that match the current retrieve profile that have yet to be retrieved from the database) then all unfetched occurrences will be retrieved. NOTE THAT FOR LARGE DATABASE TABLES THIS COULD RESULT IN A SIGNIFICANT DELAY.
TRANSACTION=TRUE	Open up another database path for the child process, thus avoiding any conflicts. NOTE THAT THIS WILL RESULT IN MULTIPLE DATABASE OPENS FOR A SINGLE USER.
RELEASE=<entity>	Drop the hitlist at the current point for the named entity. If the current hitlist is incomplete then all unfetched occurrences will be dropped and will no longer be available in the current operation - the user will have to perform a <clear> followed by a <retrieve> in order to build a new hitlist.

The last option uses the **release** command, which causes all occurrences that have been retrieved from the database to have **$dbocc** changed from **true** to **false**.

Another way to avoid this conflict between parent and child formis to use a different database entity. Two options are available:

a) Use an entity which is defined as a View within the database.

b) Create an exact copy of the target entity (**not** a subtype) for use in the List form, then in the assignment file redirect this copy to the original enrity name. In this way UNIFACE will think that you are using different physical entities and there will be no conflict in the hitlists between the parent and child forms. The fact that the two entity names refer to a single physical database table is something that is only known to, and handled by, the database engine. Note that using a subtype will not have the same effect as all physical access is automatically directed to the supertype.

4.19 AUTOMATIC RETRIEVE IN LIST FORMS

There are some forms where the dialog type is LIST that allow the user to enter their own selection criteria using the profile area at the top of the screen. Depending on user requirements these can be configured to perform an automatic retrieve upon initial entry. This option can be set by using the Extra Parameter field for the transaction definition on the Menu database. This has the following options:

$auto_retrieve$=N (default) No automatic retrieve. Upon initial entry the user will be presented with an empty screen.

$auto_retrieve$=Y Upon initial entry perform an automatic retrieve. As no profile has been defined yet this will cause all entries on that database table to be selected.

Whenever the function is selected the contents of this field is added to **$$params**, which is the single parameter that is passed to every function that is selected via a menu screen. This is processed in the <exec> trigger of that transaction in the following manner:

```
params
      string      $params$        : IN
endparams

getlistitems/id/local $params$            ; copy to local variables

clear/e "<MAIN>"
clear/e "<RETRIEVE_PROFILE>"

if ($auto_retrieve$)                      ; is automatic retrieve turned on?
      call LP_RETRIEVE                    ; yes
endif

curocc_video/inner "<MAIN>","DEF"         ; set occurrence highlighting

edit <FIRST_FIELD.<RETRIEVE_PROFILE>>
```

The value for **$auto_retrieve$** defined in Extra Parameters is then loaded into a local variable of the same name (which must be predefined with type = boolean). The program can then take the necessary action according to the value defined in this field.

NOTE: The contents of Extra Parameters on the Menu database is not restricted to the **$auto_retrieve$** option. Any number of values can be entered, and will be made available to the program in the same manner. Any values not expected by the program will be ignored.

4.20 AUTOMATIC REFRESH OF CHILD INSTANCES

There are two possible scenarios for dealing with relationships in forms where the parent form shows multiple occurrences and the child form shows a single occurrence which was selected in the parent form. In the following example the parent form is of type LIST, and the child form is of type DISPLAY:-

(1) Select occurrence 1 in form LIST, activate a new instance of DISPLAY for this occurrence.
Select occurrence n in form LIST, activate a new instance of DISPLAY for this occurrence.

This provides a separate instance of DISPLAY for each occurrence in LIST.

(2) Select occurrence 1 in form LIST, activate a new instance of DISPLAY for this occurrence.
Select occurrence n in form LIST, contents of DISPLAY changes accordingly.

Only one instance of DISPLAY is created, but its contents changes in line with each different occurrence selected in LIST.

Either one of these options can be specified as the default by the relevant setting in the assignment file. This can be changed at any point during the application by selecting the CHILDREN option on the pulldown menu, then clicking on item REFRESH. This is a “toggle” switch which is either ON or OFF. When it is ON a tick is shown in front of the item name, and the automatic refresh feature is ON. If it is OFF (no tick) then this feature is turned off.

To take advantage of this feature the following code should be inserted into the <occurrence gets focus> trigger of the main entity in the parent form:

```
if ($$refresh_children & !$empty)
        call LP_PRIMARY_KEY
        call REFRESH_CHILDREN
endif
```

There is another item on this pulldown menu called DELETE - this signifies that the current form has children which can be deleted just by selecting this item. This is an alternative to deleting each child instance individually.

4.21 APPLICATION STARTUP AND CLOSEDOWN

4.21.1 Application Startup

It is possible to perform some initial processing when the first form for an application is activated during any session by amending the contents of your application's INIT_PROC. It should contain statements as in the following example:

```
$variation = "<application mnemonic>"
if (!$$first_time_flag)           ; execute once only
    $$first_time_flag = 1         ; done!
    call SAVE_VARIATION($variation)
    <<...other initial processing...>>
endif
```

Additonal statements can be inserted within this **if/endif** clause as required, including the activation of another form or service component. Note that the use of **$$first_time_flag** ensures that this processing is performed only once per session.

Do not remove the call to the SAVE_VARIATION proc as this will prevent any closedown processing.

4.21.2 Application Closedown

When a session terminates, either by the user logging out completely or returning to the logon screen to start another session, it is possible to perform some closedown processing for each application that was accessed during the session.

Just before the menu screen ends it will activate the closedown form for each application. The name of this form will be the application library name **($variation**) suffixed by "_CLOSE". The default processing for this form is simply to reset **$$first_time_flag** ready for the next session, but it can be amended to perform additonal processing as required.

5. TIPS AND TRICKS

5.1 ENTERING PROFILES BEFORE A RETRIEVE

There are times when it is necessary to provide the user with the opportunity to enter selection criteria before initiating a <retrieve> operation. This can cause the developer some difficuties because if the standard entity definition is used then UNIFACE will perform all its syntax checks and insist that all mandatory fields are not null. This is inconvenient if the user does not want, or is unable, to provide a value for each mandatory field.

Firstly, a simple rule - do not use a single entity definition to both enter the <retrieve> profile and display the results as you will tie yourself up in knots for the reason stated above.

The method used to get around this problem varies according to the form layout.

5.1.1 Profile and Results in the same screen

This is where the profile can be entered in one area of the screen (usually the top) and the results displayed in another, thus requiring two separate entity frames. A common mistake is to use an entity subtype for the selection criteria, but this hits the same problem with mandatory fields.

The simple solution is to define a dummy (non-database) entity in the application model called RETRIEVE_PROFILE (or just PROFILE) in which you define all the fields you desire, but make them optional. It is then a simple matter to use **putlistitems/getlistitems** to transfer the values from the dummy entity to the real entity before initiating the <retrieve> operation.

This is the method that is employed in the component template for dialog type List 1.

5.1.2 Profile in a screen of its own

This is where the profile is entered on one screen, and the results displayed in another. In this case it is possible to use the true entity definition, not a dummy, provided that you do the following:

a) In the component properties set Behaviour to ‘Limited’. This will not allow database updates to be performed, therefore will not fire the field or occurrence validation triggers.

b) Replace the standard <ON_ERROR> trigger code for all fields with the following. This will allow profile characters to be entered without rejecting them:

```
if ($error = 0126)              ; syntax error
   if ($fieldprofile) return(0)  ; allow profile characters
endif

message $text("%%$error")
return(-1)
```

This is the method that is employed in the component template for dialog type Select 1.

5.2 When Null equals Infinity

If a field is empty it is said to have a null value. However, there may be circumstances in which an empty field should be treated as if it contained the highest possible value (infinity) instead of the lowest possible value (null).

For example, an entity may contain an end date to signify when it becomes unavailable, so if it contains a date that is in the past it should not be used. If an entity has no end-date it is usual practice to display it as empty rather than as a dummy date in the future, but when a null value is encountered in the comparison **where end-date >= today** it will always be rejected as it is considered to contain a lower value instead of a higher one.

One way around this is to include the test for a null value in the comparison, as follows:-

```
read u_where (end_date >= target_date | end_date = “”)
```

Another method is to hold unspecified end-dates on the database as the highest date available, but convert them to null before being displayed on any form. This removes the possibility of making a mistake with the two-stage comparison. This requires code to convert the value from infinity to null before displaying the retrieved value, and from null to infinity before writing to the database, as in the following examples:-

a) <format> trigger for the field

```
if ($format = $date("31-12-9999")) $format = ""          ; change infinity to null
```

b) <deformat> trigger for the field

```
if ($format = "") $format = "31-12-9999"                 ; change null to infinity
```

1. Some DBMS’s may not support dates as large as 31-12-9999 - check the manual for details and adjust this value accordingly.
2. For comments on possible problems with the **u_where** statement please refer to the previous section *Use of SELECTDB.*

5.3 TESTING FOR A RANGE OF VALUES

There may be some times when you need to test for multiple values within a variable. This can be done using code such as the following:

```
If (NAME = “ALF” | NAME = “BILL” | NAME = “CHAS” | NAME = “DAVE”)
        …….
```

This can also be done with the following:

```
If (NAME = ‘(ALF)(BILL)(CHAS)(DAVE)’) & (NAME != “”)
        …….
```

Note the use of single quotes to indicate a syntax string.

5.4 HELP TO LOCATE ENTRIES IN THE MESSAGE FILE

When including some validation in a component that requires an entry from the message file it can sometimes be quite tedious looking for the particular message that you want, particularly if entries relating to the same topic are scattered among other entries. Failure to identify an existing message may result in unnecessary duplication. One method which gives the ability of being able to search through the message file by a particular topic is to put the topic name in the description field. The topic may be the field name (eg: START_DATE, END_DATE, QUANTITY) or it may be a general category (eg: SECURITY). Don’t forget to use upper case characters as this will make searching easier.

5.5 PAUSING RETRIEVES ON HIGH-CAPACITY DATABASE TABLES

To avoid the possibility of a session being ‘locked’ for a significant period of time due to the retrieval of a large number of database occurrences, it is possible to include code in the <read> trigger which will interrupt the retrieve after a certain number of entries have been read. This code is contained in global proc CHK_READ_LIMIT. It uses as its limit the value from **$$read_limit=n** from the [logicals] section in the assignment file. As each record is read a counter is incremented and compared with this limit. Each time the limit is reached the user will be presented with a dialog box requiring a YES or NO response. If NO is selected the retrieve will be terminated at that point, and all remaining entries in the hitlist will be dropped. If YES is selected the retrieve will continue until the limit is reached again.

Appendix A: WIDGETS - STANDARD ENTRIES

The following settings are taken from the .INI file. Note that each widget type can have its own logical font.

```
; Logical to physical widget mapping

fEditBox=ueditbox(font=feditbox;frame=on;3d=on;dblclk=detail;multiline=on;autoselect=on)
fNoEditBox=ueditbox(font=fnoeditbox;frame=on;3d=off;multiline=on;dblclk=detail)
fEditNumber=ueditbox(font=feditnumber;frame=on;3d=on;multiline=on;autoselect=on)
fNoEditNumber=ueditbox(font=fnoeditnumber;frame=on;3d=off;multiline=on)
fCheckBox=ucheckbox(3d=on;tristate=off)
fColumnButton=ucmdbutton(font=label;halign=left)
fComboBox=ucombobox(font=editfont;frame=on;3d=on)
fCommandButton=ucmdbutton(font=fbutton;tooltip=on)
fDropDownList=udropdownlist(font=fdrop;3d=on;forcefit=on;dynamic=on)
fDropNoEdit=udropdownlist(font=fdropnoedit;3d=off;forcefit=on;dynamic=on;entries=0)
fListBox=ulistbox(font=flist;frame=on;3d=on)
fMenuButton=ucmdbutton(font=fmenubutton;tooltip=off)
fRadioGroup=uradiogroup(font=fradio;frame=on;3d=on)
fSpinButton=uspinbutton(3d=on;frame=on;autoselect=on)
Dynalabel=ueditbox(font=Label;frame=off;3d=off;autoselect=off;dblclk=detail;multiline=on)
fTab=utab(font=ftab)
fTree=utree(font=ftree;3d=on;frame=on)
```

The “f” prefix in front of the widget name signifies a custom version. These widgets should be used in preference to the standard versions provided by UNIFACE. If a new version of the **.ini** file accompanies a new release of the product it will be necessary to copy these definitions *en bloc* to the new file.

FEDITBOX and FNOEDITBOX are for ordinary data fields.
FEDITNUMBER and FNOEDITNUMBER are for numeric fields.

The EDIT and NOEDIT variations make it easy for the user to distinguish the difference between fields that are editable and those that are display only.

The FDROPNOEDIT is for a DropDownList that is display only.

The multi-line option is needed for fields if right-alignment is required when using a proportional font (see also *Appendix E: Field Layout Templates*).

Appendix B: FONTS - STANDARD ENTRIES

The following settings are taken from the .INI file. These are used to map logical font names with physical fonts.

```
[screen]
; Canvas fonts (note: Font0 is the basic screen font)
font0=Courier New,9,regular

; Default fonts for labels, buttons, and debugger
label=Arial Narrow,10,regular
buttons=Arial,8,bold
debug=Arial,10,regular

; Logical font used by toolbar, messageline
combo=Arial,11,bold

; Logical fonts used by development environment
Editfont=Arial,8,bold
Listfont=Arial,8,bold
Buttonfont=Arial,8,bold
GFP=Arial,8,bold

; Logical fonts used by Development environment
fEditbox=Arial,8,bold
fNoEditbox=Arial,8,regular
fEditNumber=Arial,9,bold
fNoEditNumber=Arial,9,regular
fButton=Arial,8,bold
fMenuButton=Arial,10,bold
fDrop=Arial,8,bold
fDropnoedit=Arial,8,regular
fRadio=Arial,8,bold
fList=Arial,8,bold
fTab=Arial,8,regular
fTree=Arial,8,regular

[printer]
font0=Courier New,10,regular
```

Appendix C: FIELD INTERFACE TEMPLATES - STANDARD ENTRIES

NAME	DATA PACKING	LEN	COMMENTS
B	B	1	boolean (Y/N) (T/F) (0/1)
COLUMN_BUTTON	C	22	
DATE	D	8	format CCYYMMDD
DATETIME	E	14	format CCYYMMDDHHNNSS
I1	I	1	integer, range +/-127
I2	I	2	integer, range +/-32,767
I3	I	3	integer, range +/-8,388,607
I4	I	4	integer, range +/-2,147,483,647
I8	I	8	integer, range +/-92,233,720,368,547,757
MONEY	M1	8	number, range +/-922,337,203,685,477.58
N1	N	1	Number, 1 digit, no decimals
N2	N	2	number, 2 digits, no decimals
N4	N	4	number, 4 digits, no decimals
N6	N	6	number, 6 digits, no decimals
N8	N	8	number, 8 digits, no decimals
N10	N	10	number, 10 digits, no decimals
N12	N	12	number, 12 digits, no decimals
POPUP_BUTTON	C	3	
PUSHBUTTON	C	22	
SS	SS		special string, shorthand=C*
TIME	T	8	format HHMMSSTT
U_VERSION	C	1	

No templates are provided for fixed-length character strings as their sizes can vary from 1 byte up to a possible 7999 bytes - use SHORTHAND definitions instead.

Appendix D: FIELD SYNTAX TEMPLATES - STANDARD ENTRIES

NAME	MAX LEN	COMMENTS
CCYY_M	4	Century and Year, mandatory
CCYY_O	4	Century and Year, optional
COLUMN_BUTTON		Noedit, noprompt
DATE_M	11	date, mandatory
DATE_O	11	date, optional
HIDDEN		noedit, noprompt, nodisplay (widget type must be UNIFIELD)
MONEY_n_d_M	15	N digits, D decimals, mandatory, unsigned
MONEY_n_d_M_N	16	N digits, D decimals, mandatory, allow negatives
MONEY_n_d_O		N digits, D decimals, optional, unsigned
MONEY_n_d_O_N		N digits, D decimals, optional, allow negatives
N1_M/O	1	Number, 1 digit, mandatory/optional
N2_M/O	2	number, 2 digits, mandatory/optional
N4_M/O	4	number, 4 digits, mandatory/optional
N6_M/O	6	number, 6 digits, mandatory/optional
N8_M/O	8	number, 8 digits, mandatory/optional
N10_M/O	10	number, 10 digits, mandatory/optional
N12_M/O	12	number, 12 digits, mandatory/optional
NOEDIT		noedit (display only)
NOPROMPT		noedit, noprompt
PERCENT	6	
POPUP_BUTTON		noedit, noprompt
PUSHBUTTON		noedit
S_M		string (see note below), mandatory
S_M_U		string, mandatory, uppercase
S_O		string, optional
S_O_U		string, optional, uppercase
TEXT_M		text (see note below), mandatory
TEXT_O		text (see note below), optional
TIME_HM_M	5	time, format=hh:nn, mandatory
TIME_HM_O	5	time, format=hh:nn, optional
U_VERSION		noedit, noprompt, nodisplay

STRING fields: characters allowed = ASCII; delete control ON; delete text control ON

TEXT fields: characters allowed = FULL; allow bold; allow italic; allow underline

NUMERIC fields: length allows the input of the decimal point and optional sign, but not any commas. Negative values cannot be input unless the LAYOUT definition specifically includes a sign.

Appendix E: Field Layout Templates - Standard Entries

NAME	FORMAT	LEN	COMMENTS
COLUMN_BUTTON			centre aligned, not active cursor
DATE	dd-MMM-yyyy	11	
DATE_DMY	dd-MMM-yyyy	11	
DATETIME	dd-Mmm-yy hh:nn:ss	18	
MONEY_6_0	zzz,zz9	7	6 digits, no decimals, unsigned
MONEY_6_2	zzz,zz9p99	10	6 digits, 2 decimals, unsigned
MONEY_6_2_N	-zzz,zz9p99	11	6 digits, 2 decimals, allow negatives
MONEY_9_2	zzz,zzz,zz9p99	14	9 digits, 2 decimals, unsigned
MONEY_9_2_N	-zzz,zzz,zz9p99	15	9 digits, 2 decimals, allow negatives
MONEY_12_2	zzz,zzz,zzz,zz9p99	18	12 digits, 2 decimals, unsigned
MONEY_12_2_N	-zzz,zzz,zzz,zz9p99	19	12 digits, 2 decimals, allow negatives
N2	z9	2	2 digits, unsigned
N4	zzz9	4	4 digits, unsigned
N6	zzzzz9	6	6 digits, unsigned
N8	zzzzzzz9	8	8 digits, unsigned
N10	zzzzzzzzz9	10	10 digits, unsigned
N12	zzzzzzzzzzz9	12	12 digits, unsigned
NOTINV			not inverse, not active cursor (used with HIDDEN syntax)
PERCENT	Zz9p99	6	right aligned
POPUP_BUTTON		3	centre aligned, not active cursor
PUSHBUTTON		22	centre aligned, not active cursor
TIME_HM	hh:nn	5	
U_VERSION		1	

For right-alignment of numeric values there are two methods:-

1. Specify **alignment=right** in the field layout (this only works if **multiline=on** is set for the widget properties). All values will be aligned with the right-hand edge of the area painted on the form, in which case any sign character should be at the beginning of the format definition.

2. Precede the format definition with the letter "**b**" - this will replace any suppressed zeroes with spaces rather than dropping the character completely, thereby aligning from the left. This will cater for a trailing sign. AT PRESENT THIS DOES NOT WORK SATISFACTORILY WITH PROPORTIONAL FONTS AS THE WIDTH OF THE SPACE CHARACTER IS SMALLER THAN ANY DISPLAYED CHARACTER.

Appendix F: FIELD TEMPLATES - STANDARD ENTRIES

NAME	DATA TYPE	INTERFACE	SYNTAX	LAYOUT	CHARACTERISTICS	WIDGET TYPE	INITIAL VALUE
B	Boolean	B			DATABASE	fCHECKBOX	
COLUMN_BUTTON	String	COLUMN_BUTTON	COLUMN_BUTTON	COLUMN_BUTTON	BOILERPLATE	FCOLUMN_BUTTON	
DATE	Date	DATE	DATE_O	DATE	DATABASE	fEDITBOX	
DATE_FROM	Date	DATE	DATE_O	DATE	NON_DATABASE	fEDITBOX	
DATE_TO	Date	DATE	DATE_O	DATE	NON_DATABASE	fEDITBOX	
DYNALABEL	String	C20	NOEDIT	NIN	BOILERPLATE	DYNALABEL	
END_DATE	Date	DATE	DATE_O	DATE	DATABASE	fEDITBOX	
MONEY_N_D	Numeric	MONEY_N_D	MONEY_N_D_O	MONEY_N_D	DATABASE	fEDITBOX	
POPUP_BUTTON	Image	POPUP_BUTTON	POPUP_BUTTON	POPUP_BUTTON	BOILERPLATE	fCOMMANDBUTTON	^u_popup_button
POPUP_FIELD	String						
PUSHBUTTON	String	PUSHBUTTON	PUSHBUTTON	PUSHBUTTON	BOILERPLATE	fCOMMANDBUTTON	
SS	Super String	SS	TEXT_O		DATABASE	UNIFIELD	
START_DATE	Date	DATE	DATE_M	DATE	DATABASE	fEDITBOX	
TIME	Time	TIME	TIME_HM		DATABASE	fEDITBOX	
TIME_FROM	Time	TIME	TIME_HMS_O		NON_DATABASE	fEDITBOX	
TIME_TO	Time	TIME	TIME_HMS_O		NON_DATABASE	fEDITBOX	
U_VERSION	String	U_VERSION	U_VERSION	U_VERSION	DATABASE	fNOEDITBOX	
VALUE_FROM	Numeric				NON_DATABASE	fEDITNUMBER	
VALUE_TO	Numeric				NON_DATABASE	fEDITNUMBER	

MONEY_N_D - ‘N’ is the number of digits before the decimal point, and ‘D’ is the number of decimal places.

POPUP_FIELD contains sample trigger code only.

Appendix G: GLOBAL PROCEDURES - STANDARD ENTRIES

All the following Global Procedures are defined in library SYSTEM_LIBRARY.

c) Initialisation

G.1. ACT_BUTTONS - PUT LABELS INTO BUTTONS ON THE ACTION BAR

This is called within the INIT operation. It identifies all the fields on the action bar, then loads their labels from the message file using an Id of "B_" followed by the field name.

Call: call ACT_BUTTONS(action_bar)

Input params: action_bar - name of dummy entity containing action buttons

Return code: none

G.2. COL_BUTTONS - PUT LABELS INTO BUTTONS ON THE COLUMN BAR

This is called within the INIT operation. It identifies all the fields on the column bar, then loads their labels from the message file using an Id of "B_" followed by the field name. If any field already has an initial value defined then this will not be overwritten.

Call: call COL_BUTTONS(column_bar)

Input params: column_bar - name of dummy entity containing action buttons

Return code: none

G.3. FULL_PROFILE_BTN - PUT LABEL INTO FULL PROFILE BUTTON

Used in LIST forms to load the label for the *Full Profile* button. This is dimmed if the user does not have access to *select_tran*.

Call: call FULL_PROFILE_BTN(button_id, select_tran)

Input params: button_id - name of dummy entity containing action buttons
select_tran - transaction that will be activated when this button is pressed

Return code: none

G.4. GET_INIT_VALUES - OBTAIN INITIAL VALUES FOR A TRANSACTION.

This procedure is available when activating an instance without using ACTIVATE_PROC. This obtains any initial values for a transaction that have been defined on the Menu database. These values will be loaded into a new occurrence by LOAD_INIT_VALUES.

Call: call GET_INIT_VALUES($init_values$)

Input params: none

Output params: $init_values$ - associative list used by LOAD_INIT_VALUES

Return code: 0=OK <0=failure ($init_values$ contains error message)

G.5. INIT_PROC - STANDARD PROC FOR <INIT> OPERATION

This performs standard initialisation at the start of every online function (except for popup forms, which use POPUP_INIT_PROC instead). Uses **$$first_time_flag** to perform initial processing for the session. Loads the form title from the message file, using an Id of "T_" followed by the form id.

Each application should have its own version to contain application-specific settings.

Call: call INIT_PROC

Return code: none.

G.6. LOAD_INIT_VALUES - LOAD INITIAL VALUES FOR A TRANSACTION.

This is designed to be used in the <OGF> trigger of new (empty) occurrences that are about to be added to the database. It transfers the contents of **$init_values$** to the current occurrence. Keywords such as **$date** and **$time** are translated into real values.

Call: call LOAD_INIT_VALUES($init_values$)

Input params: $init_values$ - associative list obtained by GET_TRAN_DATA.

Output params: none

Return code: none.

G.7. NAV_BUTTONS - PUT LABELS INTO BUTTONS ON THE NAVIGATION BAR

This is called within the INIT operation. It identifies all the fields on the navigation bar, then loads their labels from the message file using an Id of "B_" followed by the field name. If the transaction of that name is not on the user's access list then the label will be dimmed.

Call: call NAV_BUTTONS(navigation_bar)

Input params: navigation_bar - name of dummy entity containing navigation buttons

Return code: None

G.8. SAVE_VARIATION – SAVE APPLICATION NAME IN A SESSION LIST

This is called within the INIT operation. It saves the current application library name in a list so that when the session terminates a closedown form can be activated for each applicaton that was accessed during the session.

Call: call SAVE_VARIATION($variation)

Input params: $variation - library name for the current application.

Return code: None

d) Component / Instance Processing

G.9. ACTIVATE_PROC - CREATE AND ACTIVATE A CHILD INSTANCE

Used in navigation buttons to activate a form component. Calls CREATE_INSTANCE first to validate and create the instance. The instance is activated (or re-activated if it already exists) using the specified operation name, and with a single parameter (the contents of **$$params**).

Call: call ACTIVATE_PROC

Input params:
- $$component - id of form to be activated
- $$instance - name to be used for this instance
- $$properties - properties (optional - see CREATE_INSTANCE)
- $$operation - operation name (optional - default is EXEC)
- $$params - parameter string (optional - associative list)

Return code: 0 = OK <0 = cannot activate >0 = other error

G.10. BUILD_INST_NAME - BUILD INSTANCE NAME USING SPECIFIED PATTERN

Used in CREATE_INSTANCE to construct an instance name using the pattern defined in field INSTANCE_NAME for the transaction on the Menu database. PARAMS is the same associative list that is passed from the parent form to the child being activated.

Call: call BUILD_INST_NAME(component, params, pattern, instance)

Input params:
- component - component name
- params - associative list to be passed to the instance
- pattern - to be used to construct instance name

Output params:
- instance - instance name

Return code: 0 = OK <0 = error

G.11. CHILD_PROPERTIES - MANAGE HITLIST BETWEEN PARENT AND CHILD PROCESS

Used in CREATE_INSTANCE before a child process is launched. If *input_string* contains “release=<entity>” then the current hitlist on the named entity in the parent process will be truncated. If *input_string* contains “transaction=true” this will cause a new database path to be opened for the child process, thus avoiding any conflict with the hitlist.

Call: call CHILD_PROPERTIES(input_string, properties)

Input params:
- input_string - from CHILD_PROPERTIES on table MENU_TRAN in the menu database, for the parent process.

Output params:
- properties - passed to **new_instance** command

Return code: none.

G.12. CHK_INST_NAME - REMOVE INVALID CHARACTERS FROM AN INSTANCE NAME

An instance name can only contain letters, numbers and underscores, and must not be more than 16 characters long. This is called within the CREATE_INSTANCE proc.

Call: call CHK_INST_NAME(instance_name)

Input params: instance_name - original value

Output params: instance_name - with any invalid/excess characters removed

Return code: none

G.13. CREATE_INSTANCE - CREATE A NEW INSTANCE OF A COMPONENT (SPECIAL VERSION)

This proc is designed to be used only by ACTIVATE_PROC.

1. Calls GET_TRAN_DATA using **$$component** as the transaction identity, and checks to see if this function is on the user's access list. This returns all data on that transaction.
2. Calls BUILD_INST_NAME to construct the name to be used for this instance.
3. Calls CHK_INST_NAME to ensure that the instance name is valid.
4. If an instance with this name already exists then a new one will not be created.
5. Sets **$variation** to contents of LIBRARY from transaction details (if not blank).
6. **$$properties** will be updated to include the form's dimensions if HORIZPOS and VERTPOS have been defined on the transaction details).
7. If the current form is non-modal then **$$properties** is changed to force the new form to be non-modal, thus overriding the component defaults.
8. Calls CHILD_PROPERTIES if this field is not empty.
9. Calls ADD_TO_LIST to update **$$params** if EXTRA_PARMS is not blank.
10. **$$params** will be update to include any initial values if these have been defined on the Menu database for the transaction. These will be inserted with the Id of **$init_values$** so that they can only be accessed after the contents of **$$params** is loaded into component variables.
11. Issues **new_instance**, and calls PROC_ERROR if a negative status is returned.

Call: call CREATE_INSTANCE

Input params:
- $$component - component name
- $$instance - instance name
- $$properties - properties for new instance
- $$params - parameters for new instance

Return code: 0 = OK <0 = cannot create

G.14. DELETE_CHILDREN - DELETE ALL CHILD INSTANCES OF THE CURRENT COMPONENT

If the DELETE option is chosen in the pulldown menu then all instances which are attached to the current instance (ie: its children, and all their children) will be deleted.

Call: call DELETE_CHILDREN(instance_id)

Input params: instance_id - id of instance with children to be deleted

Return code: none

G.15. LAUNCH_TAB_PAGE - CREATE & ACTIVATE AN INSTANCE FOR A TAB PAGE

This proc is referenced within the component template for tab parents. It should be included in the <value changed> trigger for the tab field itself, and in the <exec> trigger in order to activate the first tab page. NOTE: the user's access to tab pages should have been already verified with a call to CHK_TAB_ACCESS.

1. Check if the field value is a component name or an instance name – if it is an instance name then **setformfocus** on that instance and exit.
2. Call CREATE_INSTANCE to create an instance of this component.
3. Update the tab field's valrep to replace "component=label" with "instance=label".
4. Activate OUTPUT_DATA operation within current component to obtain data for this tab page.
5. Activate the tab page instance.

Call: call LAUNCH_TAB_PAGE(tab_field)

Input params: tab_field - the name of the tab field
$params$ - used by CREATE_INSTANCE

Output params: none

Return code: 0 = OK <0 = error

G.16. NEW_INST_PROC - CREATE A NEW INSTANCE OF A COMPONENT

This proc is available for general use, usually to create a service component.

1. Calls INSTANCE_NAME to ensure that **instance** contains a valid name.
2. If an instance with this name already exists then a new one will not be created.
3. Calls GET_TRAN_DATA, & inserts HORIZPOS and VERTPOS into **properties.**
4. If current form is modal then inserts "modality=modal" into **properties**.
5. Calls PROC_ERROR if the **new_instance** command returns a negative status.

Call: call NEW_INST_PROC(component, instance, properties)

Input params: component - component name
instance - name to be used for this instance
properties - optional (to override default settings)

Return code: 0 = OK <0 = cannot create

G.17. POSTMESSAGE - POST A MESSAGE TO THE PARENT OF THE CURRENT COMPONENT

Called automatically by OK_PROC, STORE_PROC, STOREQ_PROC and COMMIT_PROC after a successful database update. This will sends the contents of **$$msgid** and **$$msgdata** to the instance identified in **$$msgdst**.

Call: call POSTMESSAGE(msgdst, msgid, msgdata)

Input params: msgdst - destination (optional - default is $instanceparent)
msgid - identifier (optional, default is $componentname)
msgdata - message (a string)

Output params: msgdata - cleared

Return code: 0 = OK <0 = error

G.18. REFRESH_CHILDREN - REACTIVATE CHILDREN WITH PKEY OF CURRENT OCCURRENCE

This is included in the <ogf> trigger of the main entity in forms of dialog type LIST which contain navigation buttons. If the REFRESH option on the pulldown menu is set ON then each time a different occurrence is given focus then all child instances of that form will be automatically re-activated with the primary key of the new occurrence. This will cause the contents of the child instances to switch to those of the new occurrence.

Call: call REFRESH_CHILDREN

Input params: $$params - primary key of current occurrence

Output params: $$operation - set to “EXEC”
$$properties - cleared

Return code: 0 = OK <0 = error

e) Database Processing

G.19. COMMIT_PROC - COMMIT OUTSTANDING CHANGES TO THE DATABASE

This is to be used after the STORE_NO_COMMIT proc. A **rollback** is performed in the event of an error. If no errors are found and **$$msgdata** is not empty, the POSTMESSAGE proc is called to send a message to another component (usually the parent). Includes call to AUDIT_STOP.

Call: call COMMIT_PROC

Input params: $$msgdst - passed to POSTMESSAGE proc
$$msgid - passed to POSTMESSAGE proc
$$msgdata - passed to POSTMESSAGE proc

Return code: 0 = OK <0 = error

G.20. ROLLBACK_PROC - UNDO ANY PENDING DATABASE UPDATES

This is used with the STORE_NO_COMMIT proc if any **store** errors are found. Displays a standard "store failed" message, then issues a **rollback** to undo any database changes. Includes call to AUDIT_STOP.

Call: call ROLLBACK_PROC

Output params: $$msgdata - cleared

Return code: -1

G.21. STOREQ_PROC - UPDATE THE DATABASE, BUT QUIETLY

Same as STORE_PROC, but does not issue the "STORE OK" message. If no errors are found and **$$msgdata** is not empty, the POSTMESSAGE proc is called to send a message to another component. Includes call to AUDIT_START and AUDIT_STOP.

Call: call STOREQ_PROC

Input params: $$msgdata - passed to POSTMESSAGE proc
$$msgdst - passed to POSTMESSAGE proc
$$msgid - passed to POSTMESSAGE proc

Return code: 0 = OK <0 = error

G.22. STORE_NO_COMMIT - UPDATE THE DATABASE, BUT WITHOUT A COMMIT

This is to be used when a logical update is split across several forms instead of the usual single form. Does not contain a **commit**, but issues a **rollback** if there is a failure.

Call: call STORE_NO_COMMIT

Return code: 0 = OK <0 = error

f) Error Processing

G.23. DATA_ERROR - Pass $DataErrorContext to the Message Object.

This is used to pass the contents of **$DataErrorContext** to the Message Object. It is called from within the ON_ERROR_E and ON_ERROR_F procs.

Call: call DATA_ERROR($DataErrorContext)

Input params: $DataErrorContext - the context of the last validation error

Return code: -1

G.24. GET_MESSAGE - Retrieve and display contents of Message Object

This is used to retrieve and display any messages that have been written to the Message Object. Error/Info/Warning messages are displayed in the message line while data errors and proc errors are written to the message frame. This uses proc PRINT_LIST.

Call: call GET_MESSAGE

Return code: the message count

G.25. IGNORE_MESSAGE – Remove messages from the Message Object

This is used to remove specified messages that may have been written to the Message Object so that they are excluded from the subsequent call to GET_MESSAGE.

Call: call IGNORE_MESSAGE(messagelist)

Input params: messagelist - an indexed list of message identities

Return code: none

G.26. PRINT_LIST – Print contents of indexed or asociative list in Message frame

This is used to print the contents of the passed list in the message frame. This list may be indexed or associative. Each item in the list is output on a separate line to make it easier to read.

Call: call PRINT_LIST(list)

Input params: list - an indexed or associative list

Return code: none

G.27. PROC_ERROR - PASS $PROCERRORCONTEXT TO THE MESSAGE OBJECT.

This is used to pass the contents of **$ProcErrorContext** to the Message Object as a fatal error, along with message M_90023.

Call: call PROC_ERROR($ProcErrorContext)

Input params: $ProcErrorContext - the context of the last procedure error

Return code: -1

G.28. SET_ERROR - ADD A MESSAGE (TYPE = 'E') TO THE MESSAGE OBJECT

This is used to write an error message to the Message Object for subsequent retrieval by the GET_MESSAGE proc. The message type is set to 'E'. Optional parameters will be substituted in the message text at points signified by '%%$1', '%%$2' etc, up to '%%$5'.

Call: call SET_ERROR(MessageText)

Input params: Message Text - either a text string or
- the id of a message file entry followed by optional parameters in the format '**;**1=parm1**;**2=parm2**;**. . .**;**$prompt=<fieldname>'

Return code: 0 = OK <0 = failure with the Message Object

G.29. SET_FATAL - ADD A MESSAGE (TYPE = 'F') TO THE MESSAGE OBJECT

This is used to write an error message to the Message Object for subsequent retrieval by the GET_MESSAGE proc. The message type is set to 'F'.

Call: call SET_FATAL(MessageText)

Input params: Message Text - see SET_ERROR

Return code: 0 = OK <0 = failure with the Message Object

G.30. SET_INFO - ADD A MESSAGE (TYPE = 'I') TO THE MESSAGE OBJECT

This is used to write an error message to the Message Object for subsequent retrieval by the GET_MESSAGE proc. The message type is set to 'I'.

Call: call SET_INFO(MessageText)

Input params: Message Text - see SET_ERROR

Return code: 0 = OK <0 = failure with the Message Object

G.31. SET_WARNING - ADD A MESSAGE (TYPE = 'W') TO THE MESSAGE OBJECT

This is used to write an error message to the Message Object for subsequent retrieval by the GET_MESSAGE proc. The message type is set to 'W'.

Call: call SET_WARNING(MessageText)

Input params: Message Text - see SET_ERROR

Return code: 0 = OK <0 = failure with the Message Object

g) List/String Processing

G.32. ADD_TO_LIST - ADD CONTENTS OF ONE ASSOCIATIVE LIST TO ANOTHER

This is used to merge the contents of one associative list with another. Where an entry exists in both lists the original entry is overwritten.

Call: call ADD_TO_LIST(additions, original)

Input params: additions - values to be merged
original - original values

Output params: original - updated with contents of list1

Return code: none

G.33. ASSOC_TO_INDEXED - CONVERT AN ASSOCIATIVE LIST INTO 2 INDEXED LISTS

Call: call ASSOC_TO_INDEXED(assoc, id_list, value_list)

Input params: assoc - associative list

Output params: id_list - indexed list containing 'id' part only
value_list - indexed list containing 'value' part only

Return code: none

G.34. DROP_FROM_LIST - REMOVE ITEMS FROM AN INDEXED LIST

When a list of items is obtained from a function (such as $entinfo), but needs to be edited before subsequent processing, this proc can be used to remove named items from the list.

Call: call DROP_FROM_LIST(list, exclude_list)

Input params: list - initial list of items
exclude_list - items to be removed from the initial list

Output params: list - with items removed

Return code: none

G.35. DROP_NULL_ITEM - REMOVE ITEM WITH NULL VALUE FROM ASSOCIATIVE LIST

A list obtained by putlistitems/occ will extract all items and their values, even those which do not have values. This proc will remove such items from the list.

Call: call DROP_NULL_ITEM(list)

Input params: list - associative list

Output params: list - contains only items with non-null values

Return code: none

G.36. ENTITY_LOAD - LOAD ENTITY DATA FROM A STRING

This will take the contents of an associative list created by the ENTITY_UNLOAD proc and transfer the data into the corresponding entities of the current component.

Call: call ENTITY_LOAD(EntityData)

Input params: EntityData - associative list

Output params: none

Return code: none

G.37. ENTITY_UNLOAD - UNLOAD ENTITY DATA TO A STRING

This will unload the data from all occurrences of the named entity into an associative list. The data for each occurrence will have the id of 'entname\occno'. If any occurrence has inner entities these will be appended to the list before moving on to the next occurrence.

Call: call ENTITY_UNLOAD(EntityId, EntityData)

Input params: EntityId - name of the starting entity

Output params: EntityData - contains data in an associative list

Return code: none

G.38. EXAMINE_REPLACE - EXAMINE STRING REPLACING 'A' WITH 'B'

This proc examines a string and replaces all occurrences of delimiter A (any character string) with delimiter B (any character string). This is typically used to replace "<gold>semicolon" with another character before storing a list on the database.

Call: call EXAMINE_REPLACE(string, old, new)

Input params: string - any string
old - old delimiter
new - new delimiter

Output params: string - contents adjusted

Return code: none.

h) Popup Processing

G.39. POPUP_BTN_DTL - STANDARD <DETAIL> TRIGGER FOR POPUP BUTTONS

Inserted into the <detail> trigger of the pushbutton that activates a popup. This actually causes the <detail> trigger of the previous field (the "popup" field) to be fired.

Call: call POPUP_BTN_DTL

Input params: $$popup_field_name - see POPUP_LFLD
$$popup_field_value - see POPUP_LFLD

Return code: none

G.40. POPUP_DTL - STANDARD <DETAIL> TRIGGER FOR POPUP FIELDS

Inserted into the <detail> trigger of a field which can only be populated by the popup mechanism, and not by direct user input. Please refer to section ***4.13*** *POPUP Processing* for more details. If the current field is not empty its contents will be added to the profile area before POPUP_PROC is called to activate the specified popup form.

Call: call POPUP_DTL(popup_form)

Input params: popup_form - name of popup form
$$profile - profile to be passed to the popup form

Output params: $$profile - initialised
$$selection - primary key of user's selection (associative list)

Return code: 0 = OK <0 = error >0 = nothing selected

G.41. POPUP_INIT_PROC - STANDARD PROC FOR <INIT> OPERATION IN POPUP FORMS

Performs standard initialisation for popup forms. Loads the form title from the message file, using an id of "T_" followed by the form name.

Call: call POPUP_INIT_PROC

Return code: none.

G.42. POPUP_LFLD - STANDARD PROC FOR <LEAVE FIELD> TRIGGER IN POPUP FIELDS

Inserted into the <leave field> trigger of the popup field. If the field has changed the following processing takes place:
1. The current value is stored in global variables for use by POPUP_BTN_DTL.
2. The current entity is cleared.
3. **$occcheck** on the outer entity is set to force the <LMO> trigger.

Call: call POPUP_LFLD

Output params: $$popup_field_name - name of current field
$$popup_field_value - value in current field

Return code: 0 = OK <0 = error

G.43. POPUP_PROC - STANDARD PROC FOR THE PROCESSING OF POPUP FORMS

Called by POPUP_DTL. Will activate the specified form, then issue a **retrieve** for the occurrence selected by the user.

Call: call POPUP_PROC(popup_form, profile, selection)

Input params: popup_form - name of popup form
profile - retrieve profile (associative list - optional)

Output params: selection - primary key selected (associative list)
profile - initialised

Return code: 0 = OK <0 = error >0 = nothing selected

G.44. POPUP_PROFILE - LOAD RETRIEVE PROFILE INTO POPUP FORMS

This is used in popup forms to transfer the profile values to the corresponding fields in the specified entity before attempting to retrieve records from the database.

Call: call POPUP_PROFILE(profile, entname)

Input params: profile - associative list
entname - entity name

Return code: none

G.45. POPUP_QUIT_PROC - STANDARD PROC FOR THE <QUIT> TRIGGER IN POPUP FORMS

Signals that the user left a popup form without making a selection**.**

Call: call POPUP_QUIT_PROC

Return code: **exit(1)**, which terminates the current form.

i) Triggers

G.46. CLEAR_PROC - CLEAR CURRENT SCREEN OF ALL DATA

Can be included in the <clear> trigger, if required. Performs a **RELEASE** to disconnect the current data from the database so that it can be made available as the default for new input. If the data has already been released then it will be **CLEARed** instead.

Call: call CLEAR_PROC

Return code: none

G.47. CLOSE_PROC - CLOSE (TERMINATE) THE CURRENT FORM COMPONENT

Used by functions that do not have any database updates, therefore should be included in both the <accept> and <quit> triggers.

Call: call CLOSE_PROC

Return code: none, as the proc issues an **exit(0)**, which terminates the current form.

G.48. DISABLE - DISABLE A TRIGGER

This issues a standard “This function is disabled” message. It should be inserted into any trigger that the user may expect to use, but which has been specifically disabled.

Call: call DISABLE

Return code: -1

G.49. ERASE_PROC - ERASE (DELETE) ENTRIES FROM THE DATABASE

Can be inserted into the <erase> trigger for specified functions. Will erase all retrieved entries from the database, followed by a **commit**.

Call: call ERASE_PROC

Return code: 0 = OK 0 = error >0 = not allowed

G.50. FRGF_PROC – FORM GETS FOCUS TRIGGER

Resets $VARIATION for this form when it regains focus using the value saved in $$FORM_SETTINGS by FRLF_PROC.

Call: call FRGF_PROC

Return code: none

G.51. FRLF_PROC – FORM LOSES FOCUS TRIGGER

Saves $VARIATION for this form in $$FORM_SETTINGS when it loses focus, to be restored by FRGF_PROC.

Call: call FRLF_PROC

Return code: none

G.52. HELP_PROC - RUN THE HELP FORM FOR THE CURRENT APPLICATION

Inserted into every entity <help> trigger, and in the SHOW HELP option on the pulldown menu. This activates a form with the name "_HELP" prefixed by the contents of **$variation**.

NOTE: If an entry called **<variation>_HTMLHELP=<directory>** has been defined in the **[logicals]** section of the assignment file then this proc will look for a file with a ".HTM" or ".HTML" extension for the current component in the directory specified by **<directory>**. It will then activate the default web browser on this document using the Windows API "ShellExecute".

Call: call HELP_PROC

Input params: $componentname - name of current form
$entname - name of current entity
$fieldname - name of current field

Return code: none

G.53. LMK_PROC - STANDARD PROC FOR THE <LEAVE MODIFIED KEY> TRIGGER

Inserted into the <LEAVE MODIFIED KEY> trigger of every database entity.

Call: call LMK_PROC

Return code: 0 = OK <0 = error >0 = warning

G.54. OK_PROC - STANDARD PROC FOR THE OK BUTTON OR <ACCEPT> TRIGGER

This issues a **store** to update the database. If the **store** is successful a **commit** is issued. If the **store** fails a **rollback** is issued to undo all updates. If no errors are found and **$$msgdata** is not empty, the POSTMESSAGE proc is called to send a message to another component (usually the parent). Includes call to AUDIT_START and AUDIT_STOP.

Call: call OK_PROC

Input params: $$msgdst - passed to POSTMESSAGE proc
$$msgid - passed to POSTMESSAGE proc
$$msgdata - passed to POSTMESSAGE proc

Return code: if there is an error then **return(-1)** else **exit(0)**.

G.55. ON_ERROR_E - STANDARD PROC FOR THE <ON ERROR> TRIGGER FOR ALL ENTITIES

Inserted into every <on error> trigger for entities. Displays a message based on the contents of **$error**. Calls DATA_ERROR to process the contents of **$dataerrorcontext**.

Call: call ON_ERROR_E

Return code: -1

G.56. ON_ERROR_F - STANDARD PROC FOR THE <ON ERROR> TRIGGER FOR ALL FIELDS

Inserted into every <on error> trigger for fields. Displays a message based on the contents of **$error**. Calls DATA_ERROR to process the contents of **$dataerrorcontext**.

Call: call ON_ERROR_F

Return code: -1

G.57. PRINT_PROC - STANDARD PROC FOR THE <PRINT> TRIGGER (IF REQUIRED)

Executes the **print/ask** statement.

Call: call PRINT_PROC

Return code: 0 = OK <0 = error

G.58. QUIT_PROC - STANDARD PROC FOR THE CANCEL BUTTON OR <QUIT> TRIGGER

If there are any outstanding database changes the user will be asked confirm or cancel the quit. If confirmed a ROLLBACK will be issued to undo any changes and release any locks before terminating, otherwise the quit will be cancelled. The <quit> trigger will be fired in all child instances, and if the operation is cancelled the child instance which generated the question will be given focus.

Call: call QUIT_PROC

Return code: -1 = quit cancelled, else **exit(1)** to terminate the current form.

G.59. RETRIEVE_PROC - STANDARD PROC FOR THE <RETRIEVE> TRIGGER

Performs a **retrieve**, and issues a message if **$status** is not zero.

Call: call RETRIEVE_PROC

Return code: 0 = OK <0 = error

G.60. STORE_PROC - STANDARD PROC FOR THE <STORE> TRIGGER

Issues a **store** to update the database, followed by a **commit** or **rollback** as necessary, with the corresponding message. If no errors are found and **$$msgdata** is not empty, the POSTMESSAGE proc is called to send a message to another component. Includes call to AUDIT_START and AUDIT_STOP.

Call: call STORE_PROC

Input params: $$msgdata - passed to POSTMESSAGE proc
$$msgdst - passed to POSTMESSAGE proc
$$msgid - passed to POSTMESSAGE proc

Return code: 0 = OK <0 = error

G.61. VLDK_PROC - STANDARD PROC FOR THE <VALIDATE KEY> TRIGGER

Checks for *key not found* (if $foreign is true) and *duplicate key* (if $foreign is false). If an error is found an appropriate message is issued, and control is passed to the relevant <on_error> trigger.

Call: call VLDK_PROC

Return code: 0 = OK <0 = error

j) Validation / Verification

G.62. CHK_ITEM_ACCESS - DISABLE FIELDS WHICH ARE NOT ACCESSIBLE BY THE USER

Used in the <read> trigger of the main entity in specified forms. Checks the ITEM_ACCESS table on the MENU database for the current form to see if any item-level security has been set up for the security class of the logon user. If yes, then all the items (fields) identified as disallowed will be made invisible and inaccessible to the user (including any associated field labels). If an entry on ITEM_ACCESS is not found, or an item on the screen is not contained within the list then, by default, it will not be disabled.

Call: call CHK_ITEM_ACCESS

Input params: none

Return code: none

G.63. CHK_READ_COUNT - PAUSE DATABASE RETRIEVAL AFTER A NUMBER OF RECORDS

Used in <read> trigger to issue a warning every **$$read_limit** records, giving the user the opportunity to either terminate the retrieval, or to continue. If termination is chosen then **release/e** is used to complete the hitlist at the current point. **$$read_limit** is obtained from the logicals section in the .ASN file.

Call: call CHECK_READ_COUNT(read_count, read_limit)

Input params: read_count - usually held as $read_count$
read_limit - usually from $$read_limit

Output params: read_count - incremented for each record read

Return code: 0 = continue <0 = cancel

G.64. CHK_TAB_ACCESS – CHECK IF THE USER CAN ACCESS THE PAGES OF A TAB WIDGET

Checks that the user has been granted permission to access all the pages of a tab widget. It examines the tab field's valrep list (a series of 'component=label' pairs) and calls CHK_TRAN_ACCESSQ for each component name in the list. If access to any component has not been granted then that entry is removed from the list, which means that a tab for that component will not appear and therefore cannot be selected.

Call: call CHK_TAB_ACCESS(fieldname)

Input params: fieldname - fieldname of the tab widget

Return code: 0 = OK -1 = access denied to all pages

G.65. CHK_TRAN_ACCESS - CHECK IF THE USER CAN ACCESS A TRANSACTION

Checks that the current user has been granted permission to access a particular transaction**.** This information is held on the menu database. An appropriate message is passed to the Message Object.

Call: call CHK_TRAN_ACCESS(tran_id)

Input params: tran_id - transaction identity

Return code: 0 = OK -1 = access denied

G.66. CHK_TRAN_ACCESSQ - CHECK (QUIETLY) IF THE USER CAN ACCESS A TRANSACTION

Checks that the current user has been granted permission to access a particular transaction**.** This information is held on the menu database. No message is passed to the Message Object.

Call: call CHK_TRAN_ACCESSQ(tran_id)

Input params: tran_id - transaction identity

Return code: 0 = OK -1 = access denied

G.67. VLDF_OBJSVC - VALIDATE FIELD/ENTITY VIA AN OBJECT SERVICE

This calls the object service that has been defined for the entity and either invokes validation for the specified field or the entire occurrence if the fieldname is null. Any error messages as a result of validation failures will be displayed by a call to GET_MESSAGE.

Call: call VLDF_OBJSVC(entname, fieldname)

Input params: entname - entity name
fieldname - field name (or null)

Return code: 0 = OK -1 = validation error

k) Miscellaneous

G.68. DEFAULT_LANGUAGE – GET DEFAULT LANGUAGE FROM CONTROL FILE

Reads the MNU_CONTROL table for the default language code.

Call: call DEFAULT_LANGUAGE(language_code)

Input params: none

Output params: language_code

Return code: none

G.69. BUILD_PROC_LIST - BUILD LIST OF GLOBAL PROCEDURES

This should only be used in the form which is based on the VERSION component template.

G.70. DEBUG_PROC - DEFAULT PROC FOR THE <SWITCH KEYBOARD> TRIGGER

Used during system development to turn on the DEBUG feature. Will check against the user's security class to see if a password is required or not.

Call: call DEBUG_PROC

Return code: none

G.71. DECRYPT – DECRYPT A STRING

Decrypt a string in order to return it to a readable state. The key must be the same one that was used to originally encrypt the string.

Call: call DECRYPT(key, string)

Input params: key - any string value
string - string to be decrypted

Output params: string - decrypted string

Return code: none

G.72. ENCRYPT – ENCRYPT A STRING

Encrypt a string using the variable key – this makes the string unreadable. To reverse the process the encrypted string must be decrypted with the same key.

Call: call ENCRYPT(key, string)

Input params: key - any string value
string - string to be encrypted

Output params: string - encrypted string

Return code: none

G.73. GET_SESSION_DATA – GET SESSION DATA FROM MENU LOGON

Gets session data which was set when the user logged on to the menu system. This is obtianed from component MNU_H001 in the form of an associative list which is automatically loaded into component variables. The possible item names are:

session_id	(numeric)
session_logging	(boolean)
message_logging	(boolean)
logon_user	(string)
default_language	(string)
audit_logging	(boolean)

Call: call GET_SESSION_DATA()

Input params: none

Output params: none

Return code: 0=OK <0 = error

G.74. GET_TRAN_DATA - GET TRANSACTION DATA FROM THE MENU DATABASE

Reads the Menu database to obtain details for the specified transaction, including any initial values. Also obtains the contents of CHILD_PROPERTIES from the parent. This data is passed back as an associative list. The transaction must be accessible to the current user.

Call: call GET_TRAN_DATA(tran_id, tran_data)

Input params: tran_id - transaction identity (component name)

Output params: tran_data - associative list

Return code: 0 = OK <0 = error (tran_data contains error message)

G.75. HELP_ABOUT - RUN THE HELP ABOUT FORM FOR THE CURRENT APPLICATION

Called from the HELP ABOUT option on the pulldown menu.
This activates a form with the name "_HELPA" prefixed by the contents of **$variation**.

Call: call HELP_ABOUT

Input params:

$componentname	- name of current form
$formtitle	- title of current form
$form_version$	- version number from component variable

Return code: none

G.76. OBJSVC_CLEAR - CLEAR OCCURRENCES FROM AN OBJECT SERVICE

This will clear all occurrences from an Object Service, causing all future reads to be refreshed from the database.

Call: call OBJSVC_CLEAR(entname)

Input params: entname - name of entity to be cleared

Return code: none

G.77. PROC_VERSION - OBTAIN PROCEDURE VERSION NUMBER

This should only be used in the form which is based on the VERSION component template.

Call: call *Procedure*
call PROC_VERSION("*Procedure*",$procerrorcontext)

Input params: "*Procedure*" - the name of the procedure
$procerrorcontext - result of the call to *Procedure*

Return code: none

G.78. READ_INNER_ENT – RETRIEVE INNER ENTITIES

This will retrieve all entities that are painted within the named entity. This is normally used in the <async interrupt> trigger of List forms following the addition or modification of an occurrence.

Call: call READ_INNER_ENT(entname)

Input params: entname - the name of the starting entity

Return code: none

G.79. SOUNDEX – GENERATE A SOUNDEX KEY FROM A STRING

This will convert a name into a Soundex Key. This is usually used in a search screen as it helps to identify similar sounding names, such as MARSTON, MARSDON and MARSDEN.

The format of the soundex key is 'Xnnn' where:
- 'X' is the first letter of the name (this is not converted).
- 'nnn' are the remaining letters converted into numbers.

The rules for converting letters into numbers are as follows:
- Ignore the letters A, E, I, O, U, Y, W, and H
- For each of the remaining letters assign a numeric value:
  1 for the letters B, F, P, and V
  2 for the letters C, G, J, K, Q, S, X, and Z
  3 for the letters D and T
  4 for the letter L
  5 for the letters M and N
  6 for the letter R
- If adjacent assigned numeric values are equivalent, keep only the first occurrence.
- If there are insufficient letters to produce a result containing 3 digits then pad with zeroes.

Call: call SOUNDEX(name, soundex)

Input params: name - the name string to be converted

Output params: soundex - in the format ‘Xnnn’.

Return code: none

I) Audit Logging

G.80. AUDIT_BEFOREPROC – Take snapshot of data before it is changed

This will take a snapshot of the data for the occurrence before it is changed. This should be placed in the entity's <read> trigger. Includes call to AUDIT_EXCLUDE.

Call: call AUDIT_BEFOREPROC("entname","modelname",list,)

Input params: entname - the name of the entity
modelname - the name of the application model

Output params: list - the item name in which the data will be held

Return code: 0 = OK, <0 = error

G.81. AUDIT_AFTERPROC – Write changed data to Audit Log

This will take a snapshot of the data for the occurrence when it is written to the database. This will then post both the BEFORE and AFTER data to the audit object so an entry can be written to the audit log. This should be placed in the entity's <write> and <delete> triggers. Includes call to AUDIT_EXCLUDE.

Call: call AUDIT_BEFOREPROC("entname","modelname",list, "trigger")

Input params: entname - the name of the starting entity
modelname - the name of the application model
list - the results of AUDIT_BEFOREPROC
trigger - the trigger identity

Return code: 0 = OK, <0 = error

G.82. AUDIT_EXCLUDE – Exclude those items not to be audited

This will take the associative list produced by AUDIT_BEFOREPROC & AUDIT_AFTERPROC and remove those items which are not to be audited. These items are:

characteristic = non-database
datatype = raw
datatype = image

Call: call AUDIT_EXCLUDE(list)

Input params: list - associative list obtained from an occurrence

Output params: list - same list with items removed

Return code: none

Appendix H: GLOBAL VARIABLES - STANDARD ENTRIES

These are defined in library SYSTEM_LIBRARY.

VARIABLE NAME	DATA TYPE	DESCRIPTION
COMPONENT	S	component name (see ACTIVATE_PROC)
COUNT	N	general-purpose counter
DEBUG	B	debugging switch
ENTNAME	S	entity name
FIELDNAME	S	field name
FORMNAME	S	form name
FORM_SETTINGS	S	see procs FRGF_PROC and FRLF_PROC
INSTANCE	S	instance name (see ACTIVATE_PROC)
MENU_USER_ID	S	user identity from the logon screen
MSGDATA	S	message data (see POSTMESSAGE proc)
MSGDST	S	Message destination (see POSTMESSAGE proc)
MSGID	S	message identity (see POSTMESSAGE proc)
NAVIGATION_BUTTON	S	(reserved for future use)
OPERATION	S	operation name (see ACTIVATE_PROC)
PARAMS	S	parameter string (see ACTIVATE_PROC)
POPUP_FIELD_NAME	S	used by POPUP_LFLD and POPUP_BTN_DTL
POPUP_FIELD_VALUE	$	used by POPUP_LFLD and POPUP_BTN_DTL
PROFILE	S	retrieve profile passed to a popup form, or from a selection form to a display form (a list of entries in format “fieldname=value”)
PROPERTIES	S	component properties (see ACTIVATE_PROC)
READ_LIMIT	N	record count (see CHK_READ_COUNT proc)
REFRESH_CHILDREN	B	used by REFRESH_CHILDREN proc
SELECTION	S	primary key of occurrence selected in a popup form (a list of entries in format “fieldname=value”)
STATUS	N	holds $status
VERSION_ONLY	B	used by the form which display procedure version numbers

Appendix I: FORMAT FOR MESSAGE FILE ENTRIES

Entries in the central message file should be constructed as follows:-

B_<formname>	Button text for NAVIGATION buttons, where <formname> identifies the form which is activated by this button
B_<action>	Button text for ACTION buttons, where <action> identifies the action performed by this button eg: OK, Cancel, Close, Retrieve, Store
G_<glyphname>	Tooltip message for a glyph
H_<action>	Hint text (for action buttons)
H_<formname>	Hint text (for navigation buttons)
H_<fieldname>	Hint text (for screen fields)
L_<fieldname>	Field label, for a standard label applicable to all screens (this should be set as the default value for the field label within the Application Model)
V_<fieldname>	Valrep list associated with field <fieldname> eg: radiobuttons, dropdownlists, etc
M_nnnnn	Message (there is no distinction between errors or warnings)
M_<formname>	Message to appear inside a form
Q_nnnnn	Question (message requiring a response)
T_<formname>	Form title, where <formname> identifies the form

When inserting entries into the message file it is advised to set the description field to the name of the object (in upper case) to which the entry is associated. This will enable all entries relating to a particular object to be selected more easily.

A set of standard messages is provided for use in your application, as listed in *Appendix J:*. These will be located in library USYS. As they are used by the global procs which are located in SYSTEM_LIBRARY they should not be altered. Any additional messages required by your application should be defined within your application library.

Appendix J: GLOBAL MESSAGES - STANDARD ENTRIES

These are defined in library USYS with language USA. Substitutes or alternatives may be defined within your individual application library.

NOTE: these messages will be prefixed by one of the following in order to denote their usage.

M_ = Message (no distinction between errors, warnings and information)
Q_ = Question, response required (uses **askmess** instead of **message**)

90001	STORE failed - see Message Frame for more details
90002	STORE successful
90003	No changes found - Store not executed
90004	RETRIEVE failed - invalid key passed from popup
90005	Nothing selected from popup
90006	RETRIEVE failed - see Message Frame for more details
90007	No entries were found matching this profile
90008	This function has been disabled
90009	This action is not valid on an empty occurrence
90010	This is a database occurrence - value cannot be changed
90011	This is a database occurrence - entry cannot be deleted
90012	You must CLEAR the screen before attempting a RETRIEVE
90013	Cannot delete - subordinate entries exist on %%$entname
90014	STORE failed in form %%$instancename, status = %%$status, entity = %%$entname
90015	ERASE failed - see Message Frame for more details
90016	ERASE not allowed
90017	Erase successful
90018	Controls released - data available as default for new input
90019	No help text found for %%$1
90020	Entry not found
90021	Access to this function has been disallowed
90022	You are not allowed to switch focus to another form
90023	PROC ERROR - see message frame for details
90030	Start Date cannot be later than End Date
90031	Start Date cannot be later than End Date of first %%$1
90032	Start Date must be later than Start Date of previous entry
90033	Start Date must be later than End Date of previous entry
90034	Start Date must be 1 day after End Date of previous entry
90035	Cannot amend Start Date of first entry
90036	End Date cannot be earlier than Start Date
90037	End Date cannot be earlier than Start Date of last %%$1
90038	End Date must be earlier than Start Date of next entry
90039	End Date must be earlier than End Date of next entry
90040	End Date must be 1 day before Start Date of next entry
90041	Cannot amend End Date of last entry
90042	End Date cannot be later than End Date of %%$1
90043	FROM date cannot be later than TO date
90044	TO date cannot be earlier than FROM date
90045	TO date is invalid without a FROM date
90046	Start Date cannot be earlier than Start date of %%$1
90047	Object Service not defined - validation not performed
90048	Validate Operation not defined in Object Service - validation not performed

90049	Object Service has not been defined for entity %%$1
91000	Security Violation
91001	Access to function %%$1 has not been granted
91002	There are no transactions available on this menu
91003	Transaction %%$1 is unknown to system
91004	Transaction %%$1 has been disabled
91005	Form %%$1 cannot be activated - see message frame for details
91006	Unknown Transaction Type (%%$1)
91007	Action invalid - transaction %%$1 is not a menu
91008	Action invalid - transaction %%$1 is not online
91009	Action not allowed until Item List has been created
91010	Action not allowed - transaction %%$1 is not on a menu
91011	Only characters "A", "9" and "X" are allowed
91012	New Password cannot be identical to current Password
91013	Repeat Password Must be same as New Password
91014	Password must be at least %%$1 characters long
91015	Password format must be "%%$1"
91016	This User ID is in use - multiple logons not allowed
91017	This User has been disabled
91018	A Menu cannot be added to itself.
91019	Action invalid - this Security Class has global access switched ON
91020	Cannot change Transaction Type from MENU as menu contents exist
91021	The action you have requested is restricted. Please enter a valid password to continue.
91022	There are no tabs available on this page
91023	INSTANCE_NAME: Found '(' with no matching ')'
91024	INSTANCE_NAME: No field name found between '(' and ')'
91025	INSTANCE_NAME: No field found in $$params with name '%%$1'
91026	Retry Count has been exceeded - logon aborted
91027	No data extracted from file %%$1
91028	%%$count$ records loaded from file %%$1
91029	Data does not contain entity name
91030	Cannot create entity %%$1
91031	%%$count$ records have been processed
91032	%%$count$ records extracted from %%$1
91033	Cannot initialise file %%$1
91034	This action is only valid if Transaction Type is ONLINE
91035	$SELECT_TRAN$ has not been defined for this transaction
91036	Entry must be pre-selected if $CHANGE_ALLOWED$ = NO
91037	Action is invalid if $CHANGE_ALLOWED$ = NO
91038	Characters '{|};' not allowed in passwords
91039	
91040	Entry on %%$1 no longer exists - cannot continue
91041	Warning - more than %%$1 records found

Q90001	Changes have been detected - do you really wish to Quit ?
Q90002	Do you really wish to delete this entry ?
Q90003	Do you really wish to delete this entry (and all its subordinate entries) ?
Q90004	%%$1 records have been read - do you wish to read more ?
Q90005	Do you REALLY wish to reload data from %%$1 ?
Q90006	Do you wish to replace existing data ?

Appendix K: DIALOG TYPES - STANDARD VALUES

These dialog types are used in forms that interact with the user. Different dialog types that act upon the same object should share the same sequence number so that the "family" of forms can be readily identified.

Code	Description
A	Auxiliary (data passed as parameters, and not retrieved from the database)
C	Create/Add a single occurrence
D	Delete a selected occurrence
L	List/Browse multiple occurrences in summary form
M	Multi-Purpose (Create/Read/Update/Delete on multiple occurrences)
P	Popup (Picklist)
R	Read/Enquire a single occurrence
S	Select (enter selection criteria) prior to a List
U	Update/Modify a single occurrence

Special types are used where there is no dialog with the user. These codes appear in front of the sequence number as there is usually no connection with any "family" of forms that share the same sequence number.

Code	Description
H	Hidden (no user dialog, usually implemented as a Service component)

Appendix L: MENU BARS - STANDARD ENTRIES

These objects are defined in the USYS library.

Menu Bar	Menu	Option
MAIN_MENU	FILE	Clear Retrieve Accept Store Detail Print Write to File Fetch File Quit
	EDIT	Cut Copy Paste Add Occurrence Insert Occurrence Remove Occurrence Erase Profile Find Text Font – Underline - Bold - Italic
	VIEW	First Occurrence Prev Occurrence Next Occurrence Last Occurrence Next Frame Prev Frame Zoom Quick Zoom Panel
	APPLICATION	Sort Ascending Sort Descending Refresh Children Delete Children
	HELP	Show Help About Program Keyboard Help Message Frame Debug
HELP_MENU	FILE	(see above)

Appendix M: PANELS - STANDARD ENTRIES

These objects are defined in the USYS library.

Panel	Option
SESSION	Clear Retrieve Detail Store Accept Quit First Occurrence Prev Occurrence Next Occurrence Last Occurrence Message Frame Zoom Help
POPUP	Accept Detail Quit Message Zoom

The floating session panel is usually disabled. Options that are available to transactions are usually made available on the action bar at the bottom of the screen.

The popup panel is activated with a mouse button, and is available within all functions.

Appendix N: GLOBAL CONSTANTS - STANDARD ENTRIES

The following global constants are defined in library SYSTEM_LIBRARY:

Name	Expression	Description
FATAL_ERROR	-99999	to be set when $PROCERROR is returned with a non-zero value.

Appendix O: INCLUDE PROCS - STANDARD ENTRIES

The following include procs are defined in library STD. There are to be used in services and reports to provide local versions of global procs as these types of component cannot access any global objects. In some cases these are direct copies of the global proc, while in others the behaviour has been modified to work in self-contained components.

Name	Description
ALL	includes all these procs in a single statement
ASSOC_TO_INDEXED	convert an ASSOCIATIVE list into 2 INDEXED lists
CHK_INST_NAME	check instance name for valid characters
DATA_ERROR	process the contents of $DATAERRORCONTEXT
DECRYPT	decrypt a text string
DELETE_CHILDREN	delete instance children
DISABLE	mark a trigger as disabled
ENCRYPT	encrypt a text string
EXAMINE_REPLACE	examine STRING replacing "old" with "new"
FATAL_ERROR	tests $PROCERROR and $STATUS for a fatal error
GET_MESSAGE	empty proc for services
HELP_PROC	default proc for the <HELP> trigger
LMK_PROC	empty proc for services
LMK_TRIGGER	default code for the <LEAVE MODIFIED KEY> trigger
NEW_INST_PROC	create new instance
ON_ERROR_E	<ON ERROR> trigger for entities
ON_ERROR_F	<ON ERROR> trigger for fields
POSTMESSAGE	post a message back to the parent form
PRINT_LIST	put contents of associative list in message frame
PROC_ERROR	process the contents of $PROCERRORCONTEXT
SET_ERROR	add a message (type = 'E') to the Message Object
SET_FATAL	add a message (type = 'F') to the Message Object
SET_INFO	add a message (type = 'I') to the Message Object
SET_WARNING	add a message (type = 'W') to the Message Object
STOREQ_PROC	<STORE> trigger, quiet mode (no message)
STORE_NO_COMMIT	<STORE> without commit
VLDF_OBJSVC	validate field via an object service (to be used in <leave field> or <validate field> trigger)
VLDK_PROC	<validate key> processing
VLDK_TRIGGER	default code for the <VALIDATE KEY> trigger
VLDO_OBJSVC	validate occurrence via an object service (to be used in <leave mod occ> or <validate occ> trigger)

Appendix P: CHECKLIST FOR CREATING A NEW APPLICATION

	ITEM	TICK
1)	Decide on Application Mnemonic to be used as a prefix in all component names.	
2)	Create a Library for the application with a name which is either the same as the application name, or using the prefix for component names chosen in point (1).	
3)	Create global variable FIRST_TIME_FLAG in this library.	
4)	Copy INIT_PROC from SYSTEM_LIBRARY and customise it for the new application. The contents of **$variation** must be changed to the library name chosen in point (2).	
5)	Create an application model (see next section).	
6)	Create a HELP ABOUT screen from component template CT_HELPA.	
7)	Create a HELP screen from component template CT_HELP.	
8)	Create a HELP Text Maintenance screen from component template CT_HELPM.	
9)	Create a Data Unload screen from component template CT_UNLOAD.	
10)	Create a Data Reload screen from component template CT_RELOAD.	
11)	Create a Proc Version screen from component template CT_VERSION.	
12)	Create a closedown screen from component template CT_CLOSE1.	
13)	Create a Menu database for the Application, and begin to define transactions and create menu screens.	

Appendix Q: CHECKLIST FOR CREATING A NEW APPLICATION MODEL

	ITEM	TICK
1)	Create a dummy entity called ACTION_BAR, or better still copy the one from XAMPLE.	
2)	Create a dummy entity called NAVIGATION_BAR. Define as fields all those components that are liable to be activated from navigation buttons. Use field template PUSHBUTTON as this contains all the default settings.	
3)	Create a dummy entity called RETRIEVE_PROFILE. Define as fields all those items that will be available via this mechanism.	
4)	For each field specify a LABEL value.	
5)	Create a message file entry for each field label.	
6)	For each entity that will be used in a popup form add a dummy field for the popup button. Use field template POPUP_BUTTON.	
7)	For each entity that will be used in a popup form use field template POPUP_FIELD to define the defaults for the description field. Set the correct name for the popup form.	

Appendix R: CHECKLIST FOR CREATING A NEW UNIFACE COMPONENT

	ITEM	TICK
1)	Create component using specified component template.	
2)	Ensure Window Properties are correct.	
3)	Ensure Component Properties are correct.	
4)	In Component Properties set DESCRIPTION.	
5)	In Component Properties set TITLE.	
6)	In Component Properties set LIBRARY.	
7)	In Component Properties change COMMENTS as appropriate.	
8)	Check LOCAL CONSTANTS.	
9)	Check LOCAL VARIABLES.	
10)	Create message file entry for form title.	
11)	Create message file entry for this form's navigation button text.	
12)	Create message file entry for this form's navigation button hint text.	
13)	Check detail trigger of each navigation button used within this form.	
14)	Create transaction details on the Menu database.	
15)	Add transaction tp appropriate menu(s).	
16)	Create Help text for the component.	
17)	Create help text for each field within the component.	