OPEN ABAL

Language Reference 5.1t

Abstract

This document provides a language reference including the LTS Display Functions that have been added to the ABAL Translator and Executer to facilitate the use of the new LTS ESC sequences.

Jamie Marshall

ijm@amenesik.com

Table of Contents

Introduction

This document provides a language reference, especially concerning the new functions and features that have been added to the OPEN ABAL translator OTR64 and the OPEN ABAL runtime EXA64.

EVENT INSTRUCTIONS

ABAL EXECUTION

The new EVENT (666) instruction returns an indication of the capacity of the underlying architecture of the ABAL EXA as 64bit, 32bit or 16bit.

ABAL TRANSLATON

The new EVENT (667) instruction returns an indication of the nature of the translated program native integers as 8 bytes, 4 bytes or 2 bytes representing the 64bit, 32bit and 16bit program architectures of OPEN ABAL, ABAL 3 and ABAL 1 and 2 respectively.

ABAL PUSH POP

The new EVENT (306) allows the nature of the internal ABAL PUSH POP flag to be set or inspected during program operation. This integer value bitfield of flags may be set using the environment variable of the same name ABALPUSHPOP.

The PUSH POP flag allows control over the automated screen PUSH and POP performed around a LOADGO operation as describe by the following bit field flag operations and is especially required when screens of the different programs occupy their own individual windows.

- **1: PUSH TO FILE**, activate the PUSH POP mechanism, to a local disk file in the temporary directory, firstly a SCREEN PUSH will be performed in the calling program, prior to the launch of a LOAD GO ABAL program and then the SCREEN POP will be performed on return to the calling program.
- **2: PUSH BEFORE**, indicates that the target program of the LOADGO instruction is to perform a SCREEN POP during its initial startup procedure to retrieve the state of the screen transmitted by the caller.
- **4: POP AFTER**, indicates that the target program of the LOADGO instruction is to perform a screen PUSH to transmit the state of its screen back to the calling program.
- **8: INHIBIT PUSH COLOUR**, indicates that the current foreground and background colours are not to be pushed and restored via the PUSH POP management file.
- **16: INHIBIT PUSH POSITION**, indicates that the current column and line tabulation position values are not to be pushed and restored via the PUSH POP management file.

In absence of any explicit value being specified, by an environment variable or this event instruction, this will default to the integer value of 7, namely PUSH TO FILE, PUSH BEFORE and POP AFTER.

EXA SYS LOG

The new EVENT(668) allows the EXA SYSLOG mask to be set and retrieved, controlling the emission of SYSLOG messages and warnings in special cases of EXA operation. The following constants are defined controlling the described condition.

Name	Value	Description
SYSLOG_ASSIGN	1	A SYSLOG warning will be emitted when an ASSIGN instruction reuses a currently used ASSIGN handle. An EVENT (77) for the corresponding ASSIGN handle will inhibit this SYSLOG warning.
SYSLOG_ASSIGN_OPEN	2	A SYSLOG warning will be emitted when an ASSIGN instruction reuses a currently used ASSIGN handle and the handle is still in the OPEN state. Closure of the ASSIGN handle will inhibit this SYSLOG warning.
SYSLOG_LOAD	4	A SYSLOG warning will be emitted when a CHAIN or LOAD.GO instruction launches a secondary ABAL program.
SYSLOG_PROC	8	A SYSLOG warning will be emitted to signal incorrect values and types passed as parameters during a procedure CALL.
SYSLOG_CHILD	16	A SYSLOG warning will be emitted during a system call launched through a LOAD.GO instruction, while waiting for the CHILD process to terminate. This warning will signal the various states and conditions that may be encountered.
SYSLOG_CICO	32	A SYSLOG warning will be emitted during CICO output via PRINT instructions when the low level write operation fails to output data.

The default value is set to enable all the above SYS LOG messages. The default value can be controlled by the environment variable EXASYSLOG prior to start-up of the abal EXA runtime.

TEMPORARY MEMORY USAGE

The new EVENT (997) instruction returns the level of current usage of the temporary memory defined by #MEM.

32BIT and 64BIT INTEGER CONSTANTS

The use of 32bit integer constants, and now 64bit integer constants, had been inhibited to prevent defective dynamic libraries from encountering error due to their inability to use the larger integer types. The EVENT (998) instruction allows the INTEGER management subsystem to operate correctly preserving the natural size of all integer constants.

ALLOW STOP

The EVENT (999) may be used to GET and SET the program STOP status. This is used by the ABAL WEB SERVER known as WASP, to ensure that attached programs do not stop the WEB SERVER operation.

OTR PSEUDO CONSTANTS

This section of the documentation describes the collection of pseudo constants that are recognised and handled by the OPEN ABAL translator OTR64. These are useful in source management and maintenance and in case or logging, tracing, and reporting when errors occur during program execution.

CLASS NAME

This pseudo constant will be replaced by a STRING value containing the name of the current CLASS or by the SPACE string if no class is currently active.

```
PRINT=1:CLASS_NAME,TABV(1)
```

CLASS NUMBER

This pseudo constant will be replaced by an INTEGER value containing the NUMBER of the current CLASS or by ZERO if no class is currently active.

```
PRINT=1:CLASS_NUMBER,TABV(1)
```

OBJECT NAME

This new pseudo constant will be replaced by a STRING value containing the name of the current OBJECT or by the SPACE string if no object is currently active.

```
PRINT=1:OBJECT_NAME,TABV(1)
```

OBJECT NUMBER

This pseudo constant will be replaced by an INTEGER value containing the NUMBER of the current OBJECT or by ZERO if no object is currently active.

```
PRINT=1:OBJECT_NUMBER,TABV(1)
```

METHOD NAME

This new pseudo constant will be replaced by a STRING value containing the name of the current CLASS METHOD or by the SPACE string if no method is currently active.

```
PRINT=1:METHOD_NAME,TABV(1)
```

MODULE NAME

This new pseudo constant will be replaced by a STRING value containing the value specified as the name of the current MODULE, PROGRAM or LIBRARY translation production unit.

```
PRINT=1:MODULE_NAME,TABV(1)
```

SEGMENT NAME

This new pseudo constant will be replaced by a STRING value containing either the name of the current SEGMENT under translation, or its NUMBER.

```
PRINT=1:SEGMENT_NAME,TABV(1)
```

PROCEDURE NAME

This new pseudo constant will be replaced by a STRING value containing the name of the current PROCEDURE under translation.

```
PRINT=1:PROCEDURE_NAME,TABV(1)
```

#FILE

This pseudo constant will be replaced by a STRING value containing the name of the source FILE currently being translated.

```
PRINT=1:#FILE,TABV(1)
```

#LINE

This pseudo constant will be replaced by a STRING value containing the LINE number of the source file currently being translated.

```
PRINT=1:#LINE,TABV(1)
```

#DATE

This pseudo constant will be replaced by a STRING value containing the DATE at which the source file currently was being translated.

```
PRINT=1:#DATE,TABV(1)
```

#TIME

This pseudo constant will be replaced by a STRING value containing the TIME at which the source file currently was being translated.

```
PRINT=1:#TIME,TABV(1)
```

#TRUE

This pseudo constant will be replaced by an INTEGER value of 1.

```
PRINT=1:#TRUE,TABV(1)
```

#FALSE

This pseudo constant will be replaced by an INTEGER value of 0.

```
PRINT=1:#FALSE,TABV(1)
```

#WORDSIZE

This pseudo constant will be replaced by an INTEGER value equal to the ABAL WORD SIZE of the translator, 2 bytes for 16bit, 4 bytes for 32bit and 8 bytes for 64bit architecture.

```
PRINT=1:#WORDSIZE,TABV(1)
```

#PTRSIZE

This pseudo constant will be replaced by an INTEGER value of equal to the ABAL PTR SIZE of the translator, 5 bytes for 16bit and 32bit and 9 bytes for 64bit architecture.

```
PRINT=1:#PTRSIZE,TABV(1)
```

PRINT INSTRUCTIONS

EXTENDED RGB PAINT

The standard PAINT instruction, of the PRINT and ASK instructions, now accepts STRING expressions in addition to the customary integer expressions, as its parameters. When a STRING parameter is encountered it should respect the extended RGB colour descriptions, of one of the following forms:

- International CSS standard colour name (red, blue, black, white)
- A HASH prefixed 3-byte RGB HEX digit string (#80FF80)
- A standard CSS RGB (integer, integer, integer) clause.
- A standard CSS RGBA (integer, integer, integer, float) clause

CICO Screen Functions

Four new functions have been added allowing the different pieces of residual CICO screen map information to be returned to the applications, the foreground and background colour, the text attribute, and the text character code.

The following program example demonstrates the use of these functions:

```
program "Screen Functions"
dcl     c%,l%
dcl     f%,b%,a%,t%
segment 0
        For l = 1 to conf(1)
                For c = 1 to conf(2)-1
                        Print=1:Tab(c,l),Paint(Mod(c,15),Mod(l,15)),Chr$(Mod(c+l,26)+65)
                Next c
        Next l
        For l = 1 to conf(1)
                For c = 1 to conf(2)-1
                          print=1:Tab(c,l),Atb(Screen.Atb(c,l)), ‘
                                           Paint(Screen.Foreground(c,l), ‘
                                                 Screen.Background(c,l)), ‘
                                           Chr$(Screen.Chr(c,l))
                Next c
        Next l
eseg 0
end
```

Screen.Foreground

Description

This instruction will return the integer value of the screen foreground colour at the specified column and line position.

Syntax

```
% Screen.Foreground(column%, line%)
```

Parameters

Column

The integer value of this parameter, between 1 and the limit indicated by CONF(2), will determine the text column position of the associated line position from the foreground colour will be retrieved and returned.

Line

The integer value of this parameter, between 1 and the limit indicated by CONF(1), will determine the text line position of the associated column position from the foreground colour will be retrieved and returned.

Screen.Background

Description

This instruction will return the integer value of the screen background colour at the specified column and line position.

Syntax

```
% Screen.Background(column%, line%)
```

Parameters

Column

The integer value of this parameter, between 1 and the limit indicated by CONF(2), will determine the text column position of the associated line position from the background colour will be retrieved and returned.

Line

The integer value of this parameter, between 1 and the limit indicated by CONF(1), will determine the text line position of the associated column position from the background colour will be retrieved and returned.

Screen.Atb

Description

This instruction will return the integer value of the screen text attribute at the specified column and line position.

Syntax

```
% Screen.Atb(column%, line%)
```

Parameters

Column

The integer value of this parameter, between 1 and the limit indicated by CONF(2), will determine the text column position of the associated line position from the text attribute will be retrieved and returned.

Line

The integer value of this parameter, between 1 and the limit indicated by CONF(1), will determine the text line position of the associated column position from the text attribute will be retrieved and returned.

Screen.Character

Description

This instruction will return the integer value of the screen text character code at the specified column and line position.

Syntax

```
% Screen.Character(column%, line%)
```

Parameters

Column

The integer value of this parameter, between 1 and the limit indicated by CONF(2), will determine the text column position of the associated line position from the text character code will be retrieved and returned.

Line

The integer value of this parameter, between 1 and the limit indicated by CONF(1), will determine the text line position of the associated column position from the text character code will be retrieved and returned.

LTS Display Functions

The new LTS (Lightweight Terminal Services) provides a standard CICO Terminal Emulation in a web browser environment and offers a collection of simple graphical and colour manipulation instructions that work in collaboration with the standard TEXT and COLOUR planes. in to facilitate the use of the new LTS ESC sequences for graphic display operations. Each of the following functions performs the equivalent of the corresponding LTS ESC sequence, but only if the current CICO parameter file indicates that the new LTS extension functions are available. Otherwise, the instructions will be silently ignored.

Display Label

Description

This instruction will display a text label within the defined bounding box at the indicted curser position.

Syntax

```
DisplayLabel( column%, line%, columns%, lines%, height%, font$, align$, fg$, bg$, message$ )
```

Parameters

Column

The integer value of this parameter, between 1 and the limit indicated by CONF(2), will determine the text column position from which the label will be displayed. If this value is ZERO, or if the value of the line parameter is ZERO then no positioning will be performed otherwise the corresponding ESC f sequence, as used by the PRINT TAB instruction, will be issued.

Line

The integer value of this parameter, between 1 and the limit indicated by CONF (1), will determine the text line position from which the label will be displayed. If this value is ZERO, or if the value of the column parameter is ZERO then no positioning will be performed otherwise the corresponding ESC f sequence, as used by the PRINT TAB instruction, will be issued.

Columns

The integer value of this parameter, between 1 and the limit indicated by CONF (2), will determine the width in text columns of the bounding box within which the label will be displayed. If this value is ZERO, or if the value of the ***lines*** parameter is ZERO then the label will not be displayed.

Lines

The integer value of this parameter, between 1 and the limit indicated by CONF (2), will determine the height in text rows of the bounding box within which the label will be displayed. If this value is ZERO, or if the value of the ***columns*** parameter is ZERO then the label will not be displayed.

Height

The integer value of this parameter, a reasonable font height value, will determine the ratio of the label compared to the current font height as calculated for the graphical output window.

Font

The string value of this parameter should provide a valid font family name.

Align

The string value of this parameter, from the following set of values (“R”, “L”,”C”), will determine the alignment of the text of the label within the bounding box.

Fg

The string value of this parameter should provide a valid colour description to be used as the foreground or text colour of the label.

Bg

The string value of this parameter should provide a valid colour description to be used as the background or fill colour of the label.

Message

The string value of this parameter will be right trimmed and will provide the text of the label to be displayed.

Example

Display of a page title label with Arial font and blue text on white background.

```
DisplayLabel( 1,1, Conf(2), 1, 16,”Arial”, “C”, ”blue”, “white”, ”Example of Title Label” )
```

Display Image

Description

This instruction will display an image file within the defined bounding box at the indicted curser position.

Syntax

```
DisplayImage( column%, line%, columns%, lines%, option%, url$ )
```

Parameters

Column

The integer value of this parameter, between 1 and the limit indicated by CONF(2), will determine the text column position from which the image will be displayed. If this value is ZERO, or if the value of the line parameter is ZERO then no positioning will be performed otherwise the corresponding ESC f sequence, as used by the PRINT TAB instruction, will be issued.

Line

The integer value of this parameter, between 1 and the limit indicated by CONF(1), will determine the text line position from which the image will be displayed. If this value is ZERO, or if the value of the column parameter is ZERO then no positioning will be performed otherwise the corresponding ESC f sequence, as used by the PRINT TAB instruction, will be issued.

Columns

The integer value of this parameter, between 1 and the limit indicated by CONF(2), will determine the width in text columns of the bounding box within which the image will be displayed. If this value is ZERO, or if the value of the ***lines*** parameter is ZERO then the image will not be displayed.

Lines

The integer value of this parameter, between 1 and the limit indicated by CONF(2), will determine the height in text rows of the bounding box within which the image will be displayed. If this value is ZERO, or if the value of the ***columns*** parameter is ZERO then the image will not be displayed.

Option

The integer value of this parameter will provide supplementary display options for the image. Current this should be set to ZERO (0).

Url

The string value of this parameter provides the URL from which the image to be displayed will be loaded.

Example

Display of a full screen background image.

```
DisplayImage( 1,1, Conf(2), Conf(1), 0, “https://www.amenesik.com/background.png” )
```

Display Video

Description

This instruction will display a video within the defined bounding box at the indicted curser position.

Syntax

```
DisplayVideo( column%, line%, columns%, lines%, option%, url$ )
```

Parameters

Column

The integer value of this parameter, between 1 and the limit indicated by CONF(2), will determine the text column position from which the video will be displayed. If this value is ZERO, or if the value of the line parameter is ZERO then no positioning will be performed otherwise the corresponding ESC f sequence, as used by the PRINT TAB instruction, will be issued.

Line

The integer value of this parameter, between 1 and the limit indicated by CONF(1), will determine the text line position from which the video will be displayed. If this value is ZERO, or if the value of the column parameter is ZERO then no positioning will be performed otherwise the corresponding ESC f sequence, as used by the PRINT TAB instruction, will be issued.

Columns

The integer value of this parameter, between 1 and the limit indicated by CONF(2), will determine the width in text columns of the bounding box within which the video will be displayed. If this value is ZERO, or if the value of the ***lines*** parameter is ZERO then the video will not be displayed.

Lines

The integer value of this parameter, between 1 and the limit indicated by CONF(2), will determine the height in text rows of the bounding box within which the video will be displayed. If this value is ZERO, or if the value of the ***columns*** parameter is ZERO then the video will not be displayed.

Option

The integer value of this parameter will provide the following supplementary, combinable display options for the video.

1) Auto play: when this bit is set the video will start to play when loaded
2) Loop: when this bit is set the video will loop when the end is reached

Url

The string value of this parameter provides the URL from which the video to be displayed will be loaded.

Example

Display of a full screen video.

```
DisplayVideo( 1,1, Conf(2), Conf(1), 0, “https://www.amenesik.com/video.mpg” )
```

Display Pop Up

Description

This instruction will display the indicated URL in a Pop Up Window.

Syntax

```
DisplayPopUp( url$ )
```

Parameters

Url

The string value of this parameter provides the URL of the WEB item to be loaded and displayed in a Pop Up Window.

Example

Display of the LTS session console for the user guest in another pop up window.

```
DisplayPopUp( “https://www.amenesik.com:9990/openlts/v1/console/guest” )
```

Display Size

Description

This instruction will re-dimension the LTS CICO Terminal Emulation using the provided column and line count only if the values are different to the current display dimensions.

Syntax

```
DisplaySize ( columns%, lines% )
```

Parameters

Columns

The integer value of this parameter, between 1 and a reasonably large value, will determine the width in text columns of the new display screen.

Lines

The integer value of this parameter, between 1 and a reasonably large value, will determine the height in text rows of the new display screen.

Example

Re-dimension the text emulation to 132 columns by 40 lines.

```
DisplaySize( 132, 40 )
```

Display Font

Description

This instruction will select the text font family that will be used for subsequent PRINT and ASK instructions.

Syntax

```
DisplayFont( name$ )
```

Parameters

Name

The string value of this parameter will determine the font family to be used by subsequent Display Label instructions. By default, this will be the same font family as the underlying CICO text plane.

Example

Set the current Display Label font to Helvetica.

```
DisplayFont( “Helevtica” )
```

Display Fore

Description

This instruction will set the foreground text colour that will be used for subsequent PRINT and ASK instructions.

Syntax

```
DisplayFore ( colour$ )
```

Parameters

Colour

The string value of this parameter will provide a hexadecimal colour code, an RGB or RGBA expression or a valid colour name.

Example

Set the current foreground to the RGBA value.

```
DisplayFore ( “rgba(125,222,109,0.5)” )
```

Display Back

Description

This instruction will set the background text colour that will be used for subsequent PRINT and ASK instructions.

Syntax

```
DisplayBack ( colour$ )
```

Parameters

Colour

The string value of this parameter will provide a hexadecimal colour code, an RGB or RGBA expression or a valid colour name.

Example

Set the current foreground to the standard colour name “blue”.

```
DisplayBack ( “blue” )
```

Display Color

Description

This instruction allows the corresponding colour palette entry, of the LTS display emulation, to be defined with the specified colour and alpha values. The palette entry is a standard ABAL colour code between 0 and 15 inclusive.

Syntax

```
DisplayColor ( number%, red%, green%, blue%, alpha% )
```

Parameters

Number

The integer value of this parameter determines the index number of the colour palette entry that is to be redefined. This should be a value between 0 and 15 inclusive.

Red

The integer value of this parameter determines the red fraction of the RGB colour. This should be a value between 0 and 255 inclusive.

Green

The integer value of this parameter determines the green fraction of the RGB colour. This should be a value between 0 and 255 inclusive.

Blue

The integer value of this parameter determines the blue fraction of the RGB colour. This should be a value between 0 and 255 inclusive.

Alpha

The integer value of this parameter determines the degree of opacity of the colour. This should be a value between 0 and 100 inclusive where 0 represents transparent and 100 fully opaque.

Examples

Set the colour 0, to opaque black.

```
DisplayColor ( 0,0,0,100 )
```

Set the colour 0, previously opaque black, to transparent black.

```
DisplayColor ( 0,0,0,0 )
```

Display Upload

Description

This instruction allows the File Upload Window of the LTS Terminal Emulation Web page, to be displayed or to be hidden. This window allows the user to select a file from their local computer for upload to the LTS Server. The uploaded file will preserve its original name and will be stored in the sub directory of the currently authenticated user of the main LTS upload folder.

Syntax

```
DisplayUpload ( state% )
```

Parameters

State

The integer value of this parameter determines if the window is visible or invisible.

Examples

Display the Upload Window.

```
DisplayUpload ( 1 )
```

Hide the Upload window.

```
DisplayUpload ( 0 )
```

INXS - SQL Database File Access

The traditional SI, MC, and DB file access instructions of ABAL are now connected to an underlying SQL database, MYSQL, MARIADB or POSTGRESQL, via the INXS library. This renders ABAL, OPEN, allowing dynamic data exchange between ABAL applications and Third-Party applications and web servers without the need to develop complicated interface and file transfer coordination.

Configuration

The characteristics of the implicit database access, used by the traditional, non-database access methods SI and MC, is described by the collection of environment variables provided to this effect.

INXSTYPE

This environment variable allows the explicit nature of the INXS database engine to be specified. Only the values MYSQL and PGSQL are implemented in the current version of the INXSQL library interface. The default value is MYSQL.

INXSHOST

This three-field environment variable provides the host name, the port, and the verbose flag as shown below. When Transport Layer Security has been activated, the host name portion should be set to the FQDN of the host on which the Database Engine is running.

```
Export INXSHOST="localhost:3306:0"
```

INXSUSER

This environment variable provides username for the database connection as shown below:

```
Export INXSUSER="inxsql"
```

INXSPASS

This environment variable provides password credential of the username for the database connection as shown below:

```
Export INXSUSER="inxsql"
```

INXSBASE

This environment variable provides the name of the database for the connection as shown below:

```
Export INXSBASE="inxsql"
```

INXSTTL

This environment variable controls the use of TLS, Transport Layer Security, for the underlying Database Engine.

```
Export INXSTLS=On
```

INXSQLRAW

This environment variable requests that the INXS library trace all SQL requests to the specified output channel, 1 to standard output, or 2 to standard error. The value of 0 or the absence of this variable will disactivate request tracing.

```
Export INXSQLRAW=2
```

INXSQLERROR

This environment variable requests that the INXS library trace all SQL detailed Error Messages to the specified output channel, 1 to standard output, or 2 to standard error. The value of 0 or the absence of this variable will disactivate error message tracing.

```
Export INXSQLERROR=2
```

INXSQLINSERTPS

This environment variable configures INXS to use prepared statements for the ABAL INSERT Keyword when set to 1, the default value, and the use of plain text SQL queries when set to 0.

```
Export INXSQLINSERTPS=1
```

INXSQLUPDATEPS

This environment variable configures INXS to use prepared statements for the ABAL MODIF Keyword when set to 1, the default value, and the use of plain text SQL queries when set to 0.

```
Export INXSQLUPDATEPS=0
```

MYSQL

The MYSQL Database Engine is currently fully operational for use with OPEN ABAL through the INXS adapter library. The configuration should be performed to require use of Transport Layer Security, TLS, for all client server requests.

This is configured during the standard installation deployment of OPEN ABAL as shown below:

1. Modify the ***/etc/mysql/mysql.conf.d/mysqld.cnf*** file to include the following four lines at the end:

```
ssl_ca=ca.pem
ssl_cert=server-cert.pem
ssl_key=server-key.pem
require_secure_transport=ON
```

2. Copy your system certificate from the ***/home/certificates/{domain}/cert.pem*** to the ***/var/lib/mysql/server-cert.pem*** file.
3. Copy your system private key from the ***/home/certificates/{domain}/privkey.pem*** to the ***/var/lib/mysql/server-key.pem*** file.
4. Copy your certification authority chain file from the ***/home/certificates/{domain}/chain.pem*** to the ***/var/lib/mysql/ca.pem*** file.
5. Stop the MYSQL server demon

```
Service mysql stop
```

6. Start the MYSQL server demon

```
Service mysql start
```

Create File

The standard ABAL file creation instruction is to be used for the creation of database tables. It has been extended to allow the nature of SI / MC files to be set to MEMORY or NORMAL using the following syntax.

```
CFILE=handle,D=256,K=8:Next,error
```

Creates a normal Table with 256-byte data record and an 8-byte binary primary index.

```
CFILE=handle,MD=256,K=8:Next,error
```

Creates a MEMORY Table with 256-byte data record and an 8-byte binary primary index.

The usual primary key options of LK, RK and K allow the nature of the primary index to be specified. When set to K, the primary index will be BINARY, whereas the primary index will be a STRING type for the LK and RK key options.

Columns

The traditional column descriptions provided by the ABAL KEY instruction types, to ensure total compatibility with the laxist approach previously taken with Criteria, are simply binary data types, which make the use of the data rather ungainly from other applications more accustomed to the richer data types that are available to them via SQL. To allow a finer control over integration with third party application databases, the KEY types available to the ABAL programmer have been extended as shown below.

BINARY STRING

This is the traditional, and default, Alpha Numeric KEY type prevalent in most ABAL applications and described by the following KEY instruction.

```
KEY=handle,"name",length[,A]:Next,error
```

NUMERIC

This is used for the 8bit and 16bit integer types, # and %, of the ABAL language, and described by the following KEY instruction. The difference between 8bit and 16bit integers will be determined solely by the expressed length value.

```
KEY=handle,"name",length,N:Next,error
```

BCD

This is used by both the fixed and floating point, BCD Numerical variables of the ABAL language, and described by the following KEY instruction. The optional decimals indicate the number digits required after the decimal point.

```
KEY=handle,"name",length,B[,decimals]:Next,error
```

LONG

This is used for the LONG 32bit, integer variables of the ABAL language, and described by the following KEY instruction. It could have been determined from the length of an N type KEY if it were not that these are also ways of expressing a BCD type KEY with no decimal portion.

```
KEY=handle,"name",length,L:Next,error
```

HUGE

This new KEY data type is used for the HUGE 64bit, integer variables of the ABAL language, and described by the following KEY instruction.

```
KEY=handle,"name",length,H:Next,error
```

STRING

This new KEY data type is to be used for Alphanumeric String types that will be stored in the database column using UTF8 encoding.

```
KEY=handle,"name",length,S:Next,error
```

FLOAT

This new KEY data type is to be used when the database column is known to be a float value. The value received by the ABAL application will be store in a BCD variable. The length or 4 will be interpreted to represent a FLOAT value while a length of 8 will be used to represent a DOUBLE value.

```
KEY=handle,"name",length,F:Next,error
```

Query

This new instruction has been added to allow a pure SQL statement to be submitted to the database connection without first being processed by the standard "PSEUDO SQL WHERE" statement parser. This allows bulk operations such as DELETE and UPDATE to be performed without requiring a POSIT / DOWN / DELETE or POSIT / DOWN / MODIF type construction to be employed. This instruction does not allow DATA / ROW / COLUMN retrieval. The statement should be a complete SQL statement including the verb and TABLE clause.

```
INXSQUERY=handle, statement: Next, error
```

The following examples demonstrate the types of statements.

```
INXSQUERY=handle, "DELETE FROM clients where Status=0": Next, error
INXSQUERY=handle, "UPDATE clients set Status=1 WHERE Status= 0": Next, error
```

Begin

This instruction allows the ABAL application to **start** an explicit transaction on the database associated with the corresponding file handle. If the file has been assigned as an MC file, then the implicit database will be used otherwise if the file is a database file, then the assigned database will be used.

```
INXSBEGIN=handle : Next, error
```

Commit

This instruction allows the ABAL application to **close** an open transaction on the database associated with the corresponding file handle and accept any pending changes. If the file has been assigned as an MC file, then the implicit database will be used otherwise if the file is a database file, then the assigned database will be used.

```
INXSCOMMIT=handle : Next, error
```

Rollback

This instruction allows the ABAL application to **cancel** an open transaction on the database associated with the corresponding file handle and abandon any pending changes. If the file has been assigned as an MC file, then the implicit database will be used otherwise if the file is a database file, then the assigned database will be used.

```
INXSROLLBACK=handle : Next, error
```

Select

This new instruction has been added to allow an SQL WHERE statement to be submitted to the database connection, in the context of the TABLE described by the file handle, without processing by the standard "PSEUDO SQL WHERE" statement parser. This allows standard SELECT operations to be performed without requiring a POSIT / DOWN type construction to be employed. This instruction allows DATA / ROW / COLUMN retrieval. The columns to be retrieved are described by the COLUMN LIST string parameter, or expression, after the query expression containing the WHERE clause. COLUMN LIST may be a comma separated list of simple COLUMN names, functions on COLUMN names, or any valid combination. If a COLUMN value is to be returned as well as a function of the same COLUMN, then the simple COLUMN name must precede the function of the COLUMN name. In all cases, where a value returned is a function of a COLUMN name, the size and type of the result will be

the size and type of the corresponding column. This can cause integer overflow when summating short integer columns.

```
INXSSELECT=handle, statement, column_list : Next, error, records, result
```

The first example demonstrates a simple column extraction.

```
Proc ExampleOne(handle%)
Dcl     error%
Ptr     result$(1)
Dcl     counter%
Dcl     question$=1024
Dcl     columns$=1024
Dcl     rowbuffer$=128
Field=M,rowbuffer
        Dcl Name$=32
        Dcl Town$=32
        Dcl Email$=64
Field=m
Endloc
Question = “where Status=0”
Columns = “NAME, TOWN, EMAIL”
Forget result
INXSSELECT=handle, question, columns: Next, error, counter, result
While ( counter > 0 )
        Rowbuffer = result(counter)
        Print=1:Name, Town, Email, Tabv(1)
        counter = counter – 1
Wend
Remove result
EndProc
```

Second example shows an extraction of columns using a nested SELECT expression and the Pseudo Column OID representing the primary key of the record.

```
Proc    ExampleTwo(handle%)
Dcl     error%
Ptr     result$(1)
Dcl     counter%
Dcl     question$=1024
Dcl     columns$=1024
Dcl     rowbuffer$=80
Field=M,rowbuffer
        Dcl Oid$=20
        Dcl Label$=58
        Dcl Qty=12
```

```
Field=m
EndLoc
Question = “Where ID IN (Select ARTID from sales)”
Columns = “OID, LABEL, QTY”
Forget result
INXSSELECT=handle, question, columns: Next, error, counter, result
While ( counter > 0 )
        Rowbuffer = result(counter)
        Print=1:Oid, Label, Qty, Tabv(1)
        counter = counter – 1
Wend
Remove result
EndProc
```

The third example shows an extraction of a function on a column value. Notice that the simple COLUMN name must precede the MAX and MIN functions of the same column name.

```
Proc ExampleThree(handle%)
Dcl     error%
Ptr     result$(1)
Dcl     counter%
Dcl     question$=1024
Dcl     columns$=1024
Dcl     rowbuffer$=96
Field=M,rowbuffer
        Dcl Name$=32
        Dcl MaxName$=32
        Dcl MinName$=32
Field=m
Endloc
Question = “where Status=0”
Columns = “NAME, MAX(NAME), MIN(NAME)”
Forget result
INXSSELECT=handle, question, columns: Next, error, counter, result
While ( counter > 0 )
        Rowbuffer = result(counter)
        Print=1:Name, MaxName, MinName, Tabv(1)
        counter = counter – 1
Wend
Remove result
EndProc
```

In each of the preceding examples, it should be noted that the display of the results is performed in reverse order and is responsible for the apparent inversion of the ORDER BY statement. Note that the

storage memory allocated to the pointer **result** should be released using the REMOVE instruction when the selection of column values is no longer required.

Posit

This standard instruction expects a standard WHERE statement that will be processed by the traditional "PSEUDO SQL WHERE" statement parser and all COLUMN names will be checked against KEY definitions. An SQL WHERE clause will be reconstructed by INXSQL for extraction of all columns of matching rows, and submitted to the database connection, in the context of the TABLE described by the file handle of the POSIT.

```
POSIT={handle},{question},{option}:{error},{counter}
```

- The {handle} parameter provides the number of the ASSIGN table entry of the SQL TABLE for the POSIT.
- The {question} parameter provides the "PSEUDO SQL WHERE" clause or a traditional SI/MC/BD type clause (without the preceding WHERE term).
- The {option} may be (S) to indicate that an INDEX is not to be used for the selection and a sequential search will be used instead. The (U) option will signal an error 30 since it is impossible to select fields other than the specified fields within a Distinct clause. For distinct record selection the INXS SELECT function should be used instead.
- The {error} parameter references an optional standard error management vector comprising a label and an error variable.
- The {counter} will be set to 1 or 0 corresponding to the presence of 1 or more or ZERO responses.

Count

This standard instruction expects a standard WHERE statement that will be processed by the traditional "PSEUDO SQL WHERE" statement parser and all COLUMN names will be checked against KEY definitions. An SQL WHERE clause will be reconstructed by INXSQL and submitted to the database connection, in the context of the TABLE described by the file handle of the COUNT.

```
COUNT={handle},{question},{option}:{error},{counter}
```

- The {handle} parameter provides the number of the ASSIGN table entry of the SQL TABLE for the POSIT.
- The {question} parameter provides the "PSEUDO SQL WHERE" clause or a traditional SI/MC/BD type clause (without the preceding WHERE term).
- The {option} may be (U) to indicate that the COUNT should eliminate duplicate values. In this case the COLUMN names detected in the QUESTION will be used to construct explicit DISTINCT sub-terms for the COUNT clause. When (U) is not specified the COUNT(*) clause is used.
- The {error} parameter references an optional standard error management vector comprising a label and an error variable.
- The {counter} will be set to the number of records that match the selection criteria. This variable should be of type large enough to receive the resulting value.

Collect

This standard instruction expects a standard WHERE statement that will be processed by the traditional "PSEUDO SQL WHERE" statement parser and all COLUMN names will be checked against KEY definitions. An SQL WHERE clause will be reconstructed by INXSQL for extraction of the PRIMARY INDEX of matching rows, and submitted to the database connection, in the context of the TABLE

described by the file handle of the COLLECT. The PRIMARY INDEX values will be collected from the RESULT set and returned as an array of index length strings, one for each row of the result.

```
COLLECT={handle},{question }:{error},{counter},{pointer}
```

- The {handle} parameter provides the number of the ASSIGN table entry of the SQL TABLE for the POSIT.
- The {question} parameter provides the "PSEUDO SQL WHERE" clause or a traditional SI/MC/BD type clause (without the preceding WHERE term).
- The {error} parameter references an optional standard error management vector comprising a label and an error variable.
- The {counter} parameter variable will be set to indicate the number of rows in the response pointer array.
- The {pointer} parameter variable must be an ABAL PTR variable that has been defined as an ARRAY of STRING elements. The dimensions of the string variable and the number of elements will be reinitialised during the implicit CREATE {pointer} ({index_length},{row_count}) instruction performed when at least one matching row is found.

The COLLECT instruction provides a powerful alternative to the traditional COUNT/POSIT WHILE DOWN constructions that were necessary before. An example of a standard COLLECT construction can be seen in the code snippet below.

```
DCL E%, ROW:, NB:
PTR P$=1(1)
DCL BUFFER$=256
COLLECT=1,"SOLDE > 0":Next,E, NB, P
IF (( E = 0 ) AND ( NB > 0 ))
    FOR ROW = 1 to NB
        SEARCH=1,P(ROW),/FF:Next,E,BUFFER
    NEXT ROW
    REMOVE P
ENDIF
```

Note that the storage memory allocated to the pointer P should be released using the REMOVE instruction when the collection of primary indexes is no longer required.

Insert

The INSERT instruction allows new records to be created for SI, MC and DBMC files. Each call to the instruction will perform an atomic SQL INSERT statement to be sent to the remote SQL server. This mode, the non-buffered safety mode, allows multiple applications to insert records into the same database from different host locations. Two EVENT functions have been added to allow the INSERT buffering mechanisms to be activated and disabled as required by the application, on a "table to table" basis.

EVENT (68)

This instruction allows a specific ASSIGN table to requested to use BUFFERED mode for all subsequent INSERT instructions.

```
ASSIGN=1, "mytable", MC, WR: NEXT, e, buffer
OPEN=1: NEXT, e
```

```
EVENT (68) = 1
INSERT=1, primary, /01: NEXT, e, data
```

The size of the INSERT buffer will be as indicated by the corresponding GLOBAL VARIABLE of the database motor. Buffered INSERT data will be flushed to the remote database engine prior to all other MC and SI file instructions, for the same table, thus ensuring a consistent image for the application employing the INSERT buffering technique.

EVENT (69)

This instruction disables use of the BUFFERED mode for the specified ASSIGN table for all subsequent INSERT instructions.

```
INSERT=1, primary, /01: NEXT, e, data
EVENT (69) = 1
```

Buffered INSERT data, for the specified table, will be flushed to the remote database engine during execution of this instruction. If an ERROR 81 condition is raised by the underlying flush operation, then an ERROR 81 will be raised that may be caught by an ON [LOCAL] ERROR GOTO &LABEL,VARIABLE type instruction. If no ERROR trap has been positioned then the PROGRAM will be terminated.

EVENT (78)

This instruction allows the size of the INSERT BUFFER to be specified, in KB, prior to activation of INSERT buffering using the above-described EVENT (68) for an ASSIGN handle.

```
ASSIGN=1, "mytable", MC, WR: NEXT, e, buffer
OPEN=1: NEXT, e
EVENT (78) = 1000 ; specifies the size of the INSERT buffer as one thousand 1K blocks
EVENT (68) = 1    ; activates INSERT buffering for the ASSGN handle 1
INSERT=1, primary, /01: NEXT, e, data
```

If this value is set to ZERO, the default condition, then the size of the INSERT buffer will be set as indicated by the corresponding GLOBAL VARIABLE of the database motor. Buffered INSERT data will be flushed to the remote database engine prior to all other MC and SI file instructions, for the same table, thus ensuring a consistent image for the application employing the INSERT buffering technique.

EVENT (79)

This instruction allows the maximum size of a temporary MEMORY TABLE to be specified in terms of the number of ROWS. This must be set prior to the corresponding DFILE instruction for SI, MC and MCDB files, otherwise the default value of the database engine will be used.

```
EVENT (79) = 200000          ; sets the maximum row count to 200 thousand rows
CFILE=1,MD=100,LK=10:Next,e  ; creates a temporary MEMORY TABLE
```

When the limit is reached, an ERROR 1114 will be raised.

XML File Access

This section of the OPEN ABAL documentation describes the XML file interface that allows you to read and write XML formatted files. A complete example of an XML file copier program can be found in the Annexe of this document.

Assign

The standard ASSIGN instruction is used to declare the use of a file in XML mode.

```
Assign=handle,name,XML[,WR]:Next,error
```

The WR option is required if you intend to create a new file otherwise it is not required for reading files.

Open

When opening an XML file for input, the standard OPEN instruction should be used.

```
Open=handle:Next,error
```

Cfile

When opening a new XML file for writing, the CFILE instruction should be used.

```
Cfile=handle:Next,error
```

Read

The XML information can be retrieved, sequentially, using the READ instruction, with each operation returning the next element, attribute, or value in turn.

```
Read=handle,/60:Next,error, buffer, len(buffer)
```

The structure of the buffer, submitted to the READ instruction, should be defined as follows:

```
Dcl buffer$=MAXBUFSIZE
Field=m,buffer
        Dcl type%
        Dcl length%
        Dcl value$=MAXBUFSIZE-4
Field=m
```

The ‘type’ member of this structure will return a code indicating the nature of the data, corresponding to the following list:

1. Open Element. In this case the value returned will be the name of the element.
2. Attribute Name. In this case the value returned will be the name of the attribute.
3. Attribute Value. In this case the value returned will be the value of the attribute.
4. Close Element. In this case, a value returned will be the name of the element that is closing. If no value is returned, then the element did not have any content and the name is the name of the last opened element.
5. Element Text. In this case each value returns a portion of the text data of the element content.

Each subsequent call to READ will return the next piece of information in order.

The READ instruction may also be used to retrieve the XML header information, name the VERSION, the CHARSET, and an eventual STYLESHEET name, using the following read codes:

- /0061 : Returns the XML Version value
- /0062 : Returns the XML Charset value
- /0063 : Returns the XML Style information, if any is available.

The retrieval of this information should be performed after the first standard READ operation, since the XML header will not have been parsed until the first sequential read has been performed.

Write

Writing XML data is perfectly symmetrical with the XML file READ approach using the standard WRITE codes as follows.

The XML information can be written, sequentially, using the WRITE instruction, with each operation formatting on output the next element, attribute, or value in turn.

```
Write=handle,/A0:Next,error, buffer, len(buffer)
```

The structure of the buffer, submitted to the WRITE instruction, should be defined as for the READ instruction:

```
Dcl buffer$=MAXBUFSIZE
Field=m,buffer
        Dcl type%
        Dcl length%
        Dcl value$=MAXBUFSIZE-4
Field=m
```

The 'type' member of this structure should provide a code, identical to the READ instruction, indicating the nature of the data, corresponding to the following list:

1. Open Element. In this case the value to be written will be the name of the new element.
2. Attribute Name. In this case the value to be written will be the name of the attribute.
3. Attribute Value. In this case the value to be written will be the value of the last attribute.
4. Close Element. In this case, the value should be proved will be the name of the element that is closing.
5. Element Text. In this case each value delivers a portion of the text data for the element content.

Each subsequent call to WRITE will submit the next piece of information in order.

The WRITE instruction may also be used to submit the XML header information, name the VERSION, the CHARSET, and an eventual STYLESHEET name, using the following WRITE codes:

- /00A1 : Writes the XML Version value
- /00A2 : Writes the XML Charset value
- /00A3 : Writes the XML Style information.

This information can be written at any moment prior to closure of the file.

Close

When you have finished reading or writing your XML file you should call the close instruction.

```
Close=handle:Next,error
```

JSON File Access

This section of the OPEN ABAL documentation describes the new ABAL JSON Dynamic Library interface that allows you to read and write JSON formatted files directly from ABAL. A complete example of a JSON file copier program can be found in the Annexe of this document.

The contents of the dynamic library description file can be seen below.

```
version = 1
runtime = "json"
%openjson($,%)
%createjson($,%)
%readjson(%,$,%)
%writejson(%,%,$,%)
%closejson(%)
end
```

Library Definition

The ABAL program should include the ABAL JSON Dynamic Library definition file.

```
#use "abaljson.def"
```

This may be replaced, in a Object Oriented approach, by using a class derived from the json_client class which encapsulates the dynamic library interface.

Open Json

This dynamic library function allows a JSON file to be opened for input. The first parameter is the name of the file, the second parameter is the length of the file name parameter. The function returns an integer result which will be greater than zero if successful.

```
Dcl handle%
Handle = openjson(filename,len$(filename))
```

Create Json

This dynamic library function allows a JSON file to be opened for output. The first parameter is the name of the file, the second parameter is the length of the file name parameter. The function returns an integer result which will be greater than zero if successful.

```
Dcl handle%
Handle = createjson(filename,len$(filename))
```

Read Json

The JSON information can be retrieved, sequentially, using the READ instruction, with each operation returning the next element, attribute, or value in turn.

```
Type = readjson(handle,buffer, Len(buffer))
```

The buffer submitted to the READJSON instruction, should be a simple string type and sufficiently long for the reception of the largest names and values anticipated.

The ‘type’ value returns a code indicating the nature of the data, corresponding to the following list:

0. NULL. This will be returned when the end of the JSON tree has been encountered.
1. Name. In this case the value returned will be the name of the element.

2. Value. In this case the value returned will be the string of numerical value associated with the recently returned name.
3. Structure. This type will be returned to indicate the start of a structured of complex value. The value will be empty.
4. Array Element. This type will be returned to indicate the start of a Array of values or structured of complex values. The value will be empty.
5. Close. This type will be returned to indicate the closure of the current complex value, either a structure or an array.
6. Error. This type will be returned to signal an error during the JSON parsing operation.

Each subsequent call to READ will return the next piece of information in order.

Write Json

Writing JSON data is perfectly symmetrical with the JSON file READ approach using the types to signal the nature of the data being written.

The JSON information can be written, sequentially, using the JSON WRITE instruction, with each operation formatting on output the next structure, array or simple name value pair, in turn.

```
Type = writejson(handle, type, buffer, len(buffer))
```

As for the JSON READ instruction the structure of the buffer is a simple string value.

The 'type' parameter should provide a code, identical to the READ instruction, indicating the nature of the data, corresponding to the following list:

1. Name. In this case the value returned will be the name of the element.
2. Value. In this case the value returned will be the string of numerical value associated with the recently returned name.
3. Structure. This type will be returned to indicate the start of a structured of complex value. The value will be empty.
4. Array Element. This type will be returned to indicate the start of a Array of values or structured of complex values. The value will be empty.
5. Close. This type will be returned to indicate the closure of the current complex value, either a structure or an array.

Each subsequent call to WRITE will submit the next piece of information in order.

Close Json

When you have finished reading or writing your JSON file you should call the close instruction.

```
Handle = jsonclose(handle)
```

PARQUET File Access

This section of the OPEN ABAL documentation describes the new ABAL PARQUET Dynamic Library interface that allows you to read and write PARQUET formatted files directly from ABAL. A complete example of a PARQUET file writer and reader program can be found in the Annexe of this document.

The contents of the dynamic library description file can be seen below.

```
version = 1
runtime = "parquet"
%parquetinit($,%,$,%)
%parquetcreate($,%)
%parquetcolumn(%,%,$,%,%,%)
%parquetcolumns(%)
%parquetwrite(%,$,%)
%parquetflush(%)
%parquetopen($,%)
%parquetrows(%)
%parquetread(%,&,$,%)
%parquetclose(%)
%parquetselect(%,$,%)
%parquetwhere(%,$,%)
end
```

Library Definition

The ABAL program should include the ABAL JSON Dynamic Library definition file.

```
#use “abalparquet.def”
```

This may be replaced, in an Object-Oriented approach, by using a class derived from the parquet_client class which encapsulates the PARQUET dynamic library interface.

Initialisation Parquet

This dynamic library function allows configuration of operational parameters of the PARQUET library, and especially when creating new PARQUET files. The name of the option and its length should be provided by the first two parameters. The value of the option and its length should be provided by the last two parameters.

```
Dcl handle%
Handle = parquetinit(name,len$(name),value,len$(value))
```

The following options are available:

- **ROWGROUP**: The value of this option should be an integer value indicating the number of rows to be generated per row group. *The default value is 4094* which is a good compromise between speed and size. The larger the value, the more rows that will be buffered in memory during read and write operations. The smaller the value, the greater the fragmentation of the file and the more file space dedicated to the row management.
- **FRAGMENT**: The value of this option should be either “TRUE3 or “FALSE”. *By default, fragmentation is disactivated*. When fragmentation has been activated, individual columns

will be stored in individual PARQUET data files with the main PARQUET file storing the FILE METADATA.

- **COLUMN**: The string value of this option should be either "REQUIRED" or "OPTIONAL" dictating the default nature of new columns subsequently created for new tables by the library. *The default value for this option is "REQUIRED"*.
- **ENCODING**: The string value of this option should be one of the following:
  - NONE
  - PLAIN
  - RLE
  - HYBRID
  - BITPACKED

  The default value for this option is "HYBRID".

Open Parquet

This dynamic library function allows a PARQUET file to be opened for input or append. The first parameter is the name of the file, the second parameter is the length of the file name parameter. The function returns an integer result which will be greater than zero if successful.

```
Dcl handle%
Handle = parquetopen(filename,len$(filename))
```

Create Parquet

This dynamic library function allows a PARQUET file to be created for output. The first parameter is the name of the file, the second parameter is the length of the file name parameter. The function returns an integer result which will be greater than zero if successful.

```
Dcl handle%
Handle = parquetcreate(filename,len$(filename))
```

Read Parquet

This dynamic library function allows the row of data identified by the ROW parameter to be retrieved from the PARQUET file identified by the HANDLE parameter. If the ROW requested is greater than the number of rows in the file then the READ will return a ZERO result, other wise a TRUE value will be returned, and the ROW data will be stored in the buffer provided.

```
State = parquetread(handle, row, buffer, Len(buffer))
```

The buffer submitted to the PARQUETREAD instruction, should be a simple string type and sufficiently long for the reception of the largest row anticipated. If the buffer is too small, then a ZERO response will be returned but the buffer will contain the truncated data.

Parquet Rows

This dynamic library function returns the current row count of the PARQUET file identified by the HANDLE parameter. This may be a file opened for reading or for writing.

```
Row_count = parquetrows(handle)
```

Parquet Columns

This dynamic library function returns the current column count of the PARQUET file identified by the HANDLE parameter. This may be a file opened for reading or for writing.

```
Column_count = parquetcolumns(handle)
```

Parquet Column

This dynamic library function allows a new column definition to be added to a file that is being created. The complete set of columns required should be created before outputting any row data.

```
State = parquetcolumn(handle, type, name, len$(name),length, extra)
```

The type should be one of the following integer values:

1. PARQUET_INT8
2. PARQUET_INT16
3. PARQUET_INT32
4. PARQUET_INT64
5. PARQUET_FLOAT
6. PARQUET_DOUBLE
7. PARQUET_STRING

The length should match the size of the different data types and the extra value may specify the fractional digits for the numerical types. The function will return ZERO on failure or the new column count on success.

Close Parquet

This dynamic library function allows a PARQUET that was opened for input, to be closed.

```
Dcl handle%
Handle = parquetclose(handle%)
```

Flush Parquet

This dynamic library function allows a PARQUET that was opened for output, to be flushed to storage and closed.

```
Dcl handle%
Handle = parquetflush(handle%)
```

Select Parquet

This dynamic library function allows the comma separated list of collection of column names to be specified for retrieval by subsequent Parquet Read operations.

```
State = parquetselect(handle, buffer, Len(buffer))
```

The column selection will remain positioned until the file is closed or flushed.

Where Parquet

This dynamic library function allows the complex record selection expression to be specified for retrieval by subsequent Parquet Read operations.

```
State = parquetwhere(handle, buffer, Len(buffer))
```

The selection expression will remain positioned until the file is closed or flushed.

ABAL POINTERS

This section of this documentation will provide precise information about ABAL Pointers, the way they are to be used and the way they are managed during the lifetime of an ABAL program.

Pointers were added to ABAL in version 2 of the language and allow the extension of the total ABAL variable space, accessible to the ABAL PROGRAM, almost indefinitely.

Pointers were designed, from the outset, to be useable in the same way as a standard variable of the same type in all expressions and statements which do not rely on memory underflow or memory overflow size effects.

Pointers may be declared in both GLOBAL and LOCAL variables spaces. Pointers are to be declared with explicit type, length, and dimensions, where appropriate.

Pointers may be declared in FIELD redefinition expressions of both standard variables and the regions of memory to which a base pointer variable has been pointed or directed.

Pointers may be created, to point to blocks of allocated memory or may be aliased to point to existing variables or to other allocated pointer zones.

DECLARATION

The following examples shows the declaration of all types of pointer variables. In all cases the dimensions correspond to the resulting variable type. *It is not possible in ABAL to create an ARRAY of pointers variables.*

In each of the following examples, each pointer variable will occupy a fixed amount of space in the corresponding memory area. This will be 5 bytes in ABAL 2 and ABAL 3 programs (16 and 32-bit execution mode) and has been increased to 9 bytes in ABAL 64. In both cases the initial byte will contain a flag which indicates the nature of the pointer and the remaining 4 or 8 bytes will contain the actual memory address.

1 Byte Integers

```
PTR b#
PTR bb#(16)
PTR bbb#(16,16)
```

2 Byte Integers

```
PTR c%
PTR cc%(16)
PTR ccc%(16,16)
```

4 Byte Integers

```
PTR c:
PTR cc:(16)
PTR ccc:(16,16)
```

8 Byte Integers

```
PTR c&
PTR cc&(16)
PTR ccc&(16,16)
```

BCD Strings

```
PTR n
PTR nn(16)
PTR nnn(16,16)
```

```
PTR m=12
PTR mm=12(16)
PTR mmm=12(16,16)
```

ALPHA NUMERIC Strings

```
PTR s$
PTR ss$(16)
PTR sss$(16,16)
PTR t$=256
PTR tt$=256(16)
PTR ttt$=256(16,16)
```

REDEFINITION of a POINTER

This example shows the redefinition of a pointer variable with consecutive declarations of each of the different fundamental types and finally the definition of a pointer variable to a block of the same size as the base region.

```
PTR buffer$=256
FIELD=M, buffer
DCL b#
DCL c%
Dcl i:
Dcl h&
DCL n
DCL s$
PTR p$=256
```

CREATE

This instruction is used to create a newly allocated memory block that will be accessible through the associated pointer variable.

Syntax

The complete syntax of the CREATE instruction is shown below.

```
CREATE {pointer_name} [ ( {variable_length} [, {first_dimension}] [, {second_dimension}] )]
```

Where:

- The term {pointer_name} must be the name of a variable that has been declared using the PTR keyword.
- The term {variable_length} represents the resulting integer value of a constant integer expression to be used as the length of a BCD or STRING type variable.
- The term {first_dimension} represents the resulting integer value of a constant integer expression to be used as the size of the first dimension of an ARRAY type variable.
- The term {second_dimension} represents the resulting integer value of a constant integer expression to be used as the size of the second dimension of an ARRAY type variable.

Examples

```
PTR p$
CREATE p
```

This first example will allocate a memory block of fifteen characters that will be initialised with space characters and accessible using the pointer “p”.

```
PTR p$
CREATE p(1000)
```

This second example will allocate a memory block of one thousand characters that will be initialised with space characters and accessible using the pointer “p”. The declaration of the pointer will be “adapted” by the instruction create to become a pointer to a string of one thousand characters long.

```
PTR p$(5)
CREATE p
```

This third example will allocate a memory block of five times fifteen characters that will be initialised with space characters and accessible using the pointer “p”. The memory will be accessible through “p” as an array of five, fifteen-byte strings.

```
PTR p$(1)
CREATE p(100,10)
```

This fourth example will allocate a memory block of ten times one thousand characters that will be initialised with space characters and accessible using the pointer “p”. The memory will be accessible through “p” as an array of ten, thousand-byte strings. The declaration of the pointer will be “adapted” by the instruction create to become a pointer to an array of ten strings of one thousand characters long.

In each of the preceding examples where the pointer declaration has been adapted, this adaptation will only be preserved for the duration of the variable description table. Local variable tables may lose adaptations between procedure calls depending on the actual operational environment in which the program is running. Subsequently it is good practice to preserve the dimensions of pointers that have been adapted in this way, using traditional DCL type variables to store the length and array dimensions, and to explicitly set the adaptation using the ALTER instruction, provided to allow this to be performed safely, quickly, and easily.

ALTER

This instruction is provided to allow the dimensions of a pointer variable, that are stored in a local or global variable table description, to be “adapted” or “altered” to correspond to the known and preserved dimensions of the pointer as established by a CREATE instruction but may have been lost due to variable table refresh operations that may be performed during SEGMENT load and return or PROCEDURE call and exit instructions.

Syntax

The complete syntax of the CREATE instruction is shown below.

```
ALTER {pointer_name} [ ( {variable_length} [, {first_dimension}] [, {second_dimension}] )]
```

Where:

- The term {pointer_name} must be the name of a variable that has been declared using the PTR keyword.
- The term {variable_length} represents the resulting integer value of a constant integer expression to be used as the length of a BCD or STRING type variable.

- The term {first_dimension} represents the resulting integer value of a constant integer expression to be used as the size of the first dimension of an ARRAY type variable.
- The term {second_dimension} represents the resulting integer value of a constant integer expression to be used as the size of the second dimension of an ARRAY type variable.

Examples

```
PTR p$
ALTER p(1000)
```

This example corresponds to the ALTER required as mentioned in the second example of the CREATE instruction.

```
PTR p$(5)
ALTER p(1000,10)
```

This example corresponds to the ALTER required as mentioned in the fourth example of the CREATE instruction.

REMOVE

This instruction is used to remove or delete or to liberate an allocated memory block that will be subsequently no longer accessible through the associated pointer variable. The instruction is symmetrical with the preceding CREATE and ALTER instructions, allowing length and dimension information to be provided. Any length or dimension information that is provided will be used to RESET the corresponding variable description back to the provided values. Otherwise, the current alterations will be preserved for the variable descriptor.

Syntax

The complete syntax of the REMOVE instruction is shown below.

```
REMOVE {pointer_name} [ ( {variable_length} [, {first_dimension}] [, {second_dimension}] )]
```

Where:

- The term {pointer_name} must be the name of a variable that has been declared using the PTR keyword.
- The term {variable_length} represents the resulting integer value of a constant integer expression to be used to RESET the length of a BCD or STRING type variable.
- The term {first_dimension} represents the resulting integer value of a constant integer expression to be used to RESET the size of the first dimension of an ARRAY type variable.
- The term {second_dimension} represents the resulting integer value of a constant integer expression to be used to RESET the size of the second dimension of an ARRAY type variable.

Examples

The first example shows the removal of an allocated pointer with or without the reset of the variable descriptor length information.

```
PTR p$
REMOVE p
REMOVE p(15)
```

The second example shows the removal of an allocated pointer with or without the reset of the variable descriptor length and first dimension information.

```
PTR p$(1)
REMOVE p
REMOVE p(15,1)
```

FORGET

This instruction is used to reset a pointer to NULL and disconnect it from the memory to which I may be connected. Only the 5 or 9 bytes used to store the pointer are affected and reset to binary ZERO. This instruction will, under no circumstances, release the memory to which the pointer was connected.

Syntax

The complete syntax of the REMOVE instruction is shown below.

```
FORGET {pointer_name}
```

- The term {pointer_name} must be the name of a variable that has been declared using the PTR keyword.

ALIAS

This instruction allows a pointer to create to point to existing variable or created pointer destination. The nature of the source and target data variables must be compatible.

```
{pointer_name} = Alias( {source_variable} )
```

- The term {pointer_name} must be the name of a variable that has been declared using the PTR keyword.
- The term {source_variable} must be the name of the variable to which the resulting point variable is to be connected. If the source is a pointer, the target will point to the same allocated memory area. If the source is a standard variable, the target will be aliased to the address of the variable within the corresponding data storage memory, local or global.

Warning

Global pointers may be aliased to Local variables but if the scope in which the connection is established is exited then the behaviour of the global variable is unpredictable and can cause serious memory corruption errors to occur since the local variable memory will have been released and reallocated to another purpose.

VALIDPTR

This instruction returns the initial control byte of the pointer variable allowing the state of the pointer to be examined and the nature of its usage.

```
{integer_lvalue} ValidPtr( {pointer_name} )
```

- The term {pointer_name} must be the name of a variable that has been declared using the PTR keyword.
- The term {integer_lvalue} may be an integer variable affectation of a conditional expression such as IF, WHILE, REPEAT, SELECT or CASE.

The value returned by VALIDPTR will be one of the following:

Symbolic	Value	Description
NULL	/00	The pointer is not connected

ALLOCATED	/01	The target of the pointer is the result of a CREATE, ATTACH or an ALIAS of a CREATE or ATTACH.
REFERENCE	/03	The target of the pointer is a standard variable or expression resulting from an ALIAS.
PROCEDURE	/85	The pointer is an indirection to a PROCEDURE in the current PROGRAM execution context.
SEGMENT	/09	The pointer is an indirection to a SEGMENT in the current PROGRAM execution context.
DYNAMIC LIBRARY FUNCTION	/11	The pointer is an indirection to a dynamic library function and comprises both library and function identification values.

ATTACH

This instruction allows an ABAL PROGRAM to be attached to a POINTER, which when suitably redefined, allows invocation of the procedures of the program and access to its global variables. Multiple attachment instances, resulting from multiple uses of the ATTACH instruction will result in multiple program instances, each with an independent global variable image (program object container) whilst sharing the procedure indirection table and other attachment support structures. The structures behind the attachment pointer are complex and execution context dependent so care must be taken not to damage their memory.

Two alternative forms are possible, one where the ABAL PROGRAM name is provided as a string parameter, the other where the ABAL PROGRAM name is retrieved from the ASSIGN TABLE entry corresponding to the provided integer parameter.

EXAMPLE

```
PTR p$=18
FIELD=M,p
        PTR PROC mt(1)
        PTR vg
FIELD=M
;** either
ATTACH P(“program.at”)
;** or
ASSIGN=1,“program.at“
ATTACHE P(1)
```

Upon successful completion of the ATTACH operation, the pointer P will be allocated and will contain a pointer to the procedure table and a pointer to the storage memory of the global variables. The global variable pointer may be redefined accordingly to access any, and all, variables of the program.

Pointers to attached programs may be passed as parameters to procedures invoked through the procedure indirection table allowing attach programs to CALL back, or through, to other attached programs allowing a complex community of collaboration to be achieved.

CALL

This instruction allows invocation of procedure pointers, that were previously initialised by the PROC PTR instruction.

```
CALL ({procedure_pointer_name})({parameter_values})
```

- The term {procedure_pointer_name} must be the name of a variable that has been declared using the DCL PROC, or PTR PROC keywords.
- The term {parameter_values} represents the list of commas separated parameter values required to be passed to the corresponding procedure.

EXAMPLES

Invocation of the procedure pointers of the preceding example of the ATTACH instruction, is show below.

```
CALL (mt(1))(1,2,3)
CALL (mt(2))("examples")
```

DETACH

This instruction allows an ABAL PROGRAM that has previously been attached to a POINTER, to be detached from the pointer.

```
DETACH {pointer_name}
```

The attachment support structures, and procedure indirection table will be released when the last detachment instance has been released. Any remaining attached instances will be detached and released during an eventual STOP instruction, either terminal, or during PROGRAM CHAIN operation.

PROC PTR

This instruction allows a PROCEDURE pointer, such as those contained in the attachment object procedure indirection table, to be initialised to point to a procedure of the current program context, then subsequently invoked using the CALL instruction.

```
{procedure_pointer_name} = PROC PTR {procedure_name}
```

- The term {procedure_pointer_name} must be the name of a variable that has been declared using the DCL PROC, or PTR PROC keywords.
- The term {procedure_name} will be used to resolve the target PROCEDURE of the indirection.

EXAMPLE

```
PROGRAM "example"
DCL PROC example
PROC myexample() :: Endproc
SEGMENT 0
Example = PROC PTR myexample
CALL (example)()
ESEG 0
END
```

SEGMENT PTR

This instruction allows a SEGMENT pointer to be initialised, to point to a SEGMENT of the current program context, then subsequently invoked using the CALL instruction.

```
{segment_pointer_name} = SEGMENT PTR {segment_identity}
```

- The term {segment_pointer_name} must be the name of a variable that has been declared using the DCL SEGMENT, or PTR SEGMENT keywords.
- The term {segment_identity}, either a segment number or a segment name, will be used to resolve the target SEGMENT of the indirection.

EXAMPLE

```
PROGRAM « example »
DCL SEGMENT example
SEGMENT 0
Example = SEGMENT PTR 2
CALL (example)()
ESEG 0
SEGMENT 2 :: ESEG 2
END
```

USER PTR

This instruction allows a DYNAMLIC LIBRARY or USER function pointer to be initialised, to point to a DYNAMIC LIBRARY function of the current program context, then subsequently invoked using the CALL instruction.

```
{user_pointer_name} = USER PTR {dynamic_library_function_name}
```

- The term { user_pointer_name } must be the name of a variable that has been declared using the DCL USER, or PTR USER keywords.
- The term { dynamic_library_function_name } will be the name of a dynamic library function described by the corresponding definitions file used to resolve the target DYNAMIC LIBRARY FUNCTION of the indirection.

EXAMPLE

```
#user “asfun.def”
PROGRAM “example”
DCL USER example
SEGMENT 0
Example = USER PTR winit
CALL (example)()
ESEG 0
END
```

OBJECT ORIENTED ABAL

Introduction

Object Orient ABAL, or ABAL++ as it was later to become known, was originally designed to facilitate the development and management of very large-scale business applications. At that time, back in 1989, object-oriented programming, though heard of, it was seriously frowned upon by the then established community of the structured programming enthusiasts. Object Oriented concepts were never designed as a replacement for structured programming techniques, instead they were intended to complement existing methodology and algorithms with encapsulation to facilitate integration and re-useability of code. Modern day cloud computing advances have resulted in the generalized acceptance of the rationalisation of IT in terms of COMPUTE, STORAGE and NETWORK with the former embodying the concept of CODE, the second the concept of DATA and the later to represent the communication and organisational methods by which "inter" CODE and DATA relationships may be supported, performed, and managed between remote processes, machines, and devices.

This introductory section to the Object-Oriented version of ABAL describes the reasons leading to the decisions that were taken on the journey of transformation from the state of STRUCTURED programming with ABAL to the state of OBJECT programming with ABAL.

DATA

The data definition instructions of ABAL allow an application programmer to describe the data structures required by their application with a focus on, and high degree of control over, the layout of that data in the physical memory of the program, either global or local. ABAL is a member of the group of computer languages known as "Strict Type Checking" languages where all data access and operations are carefully controlled, during the compilation phase, to ensure that they are of the appropriate and anticipated type. Consequently, variables and constants are all declared (or defined) within a particular namespace, either GLOBAL or LOCAL, with a unique name and an explicit data type, where a LOCAL declaration of a particular name will be said to "hide" or "mask" a GLOBAL declaration of the same name. The combination of DATA declarations, using the ABAL keywords CONST, DCL, PTR, FIELD and FILLER allow the description of complex data structures to perfection, not only with an absolute control and understanding of the implantation of the corresponding values in the computer's physical memory, but also with a level of abstraction allowing the same data structures to be perfectly preserved and operational on any existing computer memory management hardware of BIG or LITTLE ENDIAN format and RESTRICTED or UNRESTRICTED byte-aligned access. The ABAL runtime, EXA, being responsible for all memory access operations, both fetch and store, provides the appropriate "just in time" transformations that allow this abstraction of data not only to be possible but also to be portable across both time and space.

The predecessor of ABAL, known as B.A.L., was limited in scope to just 64K of memory, for the storage of not only all data, both constants and variables alike, but also the current segment of code under execution, with a constant trade off, required to be performed by the programmer, between the number of global variables and constants of a program and the maximum size of a segment of code.

ABAL was designed to exceed these limitations, primarily within identical hardware constraints of the 64K unit, but ultimately targeting the anticipated and emerging environments that we all know today with their unconstrained access and 64bit addressing. Not only was the GLOBAL variable and constant management mechanism extended, to offer up to 64K each, but was complemented by the addition of the equivalent LOCAL variable and constant management mechanisms, equally for all SEGMENT, and the newly added PROCEDURE, code blocks.

The resulting increase in both size and scope of the potential for each program's data declarations and definitions, even within the confines of the original 64K boundary, saw an almost immediate ultra-proliferation of the total number variables and constants declared and used within a single ABAL PROGRAM, for the description of their complex business management data structures. The replication of these data structures became a very complex and time-consuming operation, requiring the appropriate renaming of the multitude of member variables comprising the logical data structure, to ensure their name-space unicity, the management of which, over time, became a major point of fragilization and a frequent source of anomalies and program mal-function.

The first extensions to ABAL, provided by ABAL++, in the form of the #STRUCT/#ENDSTRUCT and #UNION/#ENDUNION were designed, especially, to alleviate this problem, with the ABAL compiler / translator, OTR, becoming responsible for the declaration and naming of all structure member variables and with the addition of the corresponding OBJECT.MEMBER language constructions, subsequently allowing programmer access to the internal member declarations of the resulting structures. The collection of individual member variables declarations, required for the description of each structure, are performed by the translator, OTR, using the identical pseudo DCL, PTR, FIELD and FILLER instructions that would be required to achieve the same results, if they had been explicitly individually declared by the programmer. The name of each member variable being only known to the translator, is to be represented by the programmer by the appropriate instance name and member name combination.

EXAMPLE STRUCTURE

Considering the following example of an ADDRESS structure, leaving aside the sizes of the elements, a general approach could be declared using standard ABAL data declaration instructions as follows.

```
DCL ADDRESS$
FIELD=M, ADDRESS
   DCL STREET$
   DCL LOCALITY$
   DCL TOWN$
   DCL CODE$
  DCL COUNTRY$
FIELD=M
```

The addition of a second address management structure would require the replication of all variable declarations with new names, naturally very cumbersome and very prone to error.

```
DCL ADDRESS2$
FIELD=M, ADDRESS2
   DCL STREET2$
   DCL LOCALITY2$
   DCL TOWN2$
   DCL CODE2$
  DCL COUNTRY2$
FIELD=M
```

The use of a structure greatly simplifies the replication of structural data types and alleviates all eventual adjacent anomalies.

```
#STRUCT ADDRESS
   DCL STREET$
   DCL LOCALITY$
   DCL TOWN$
   DCL CODE$
  DCL COUNTRY$
#ENDSTRUCT ADDRESS
ADDRESS A
ADDRESS B
ADDRESS C
```

The members of each structure being accessible through the corresponding *"OBJECT NAME . MEMBER NAME"* construction, as shown below.

```
A.TOWN = "PARIS"
B.TOWN = "NICE"
C.TOWN = A.TOWN
```

EXAMPLE UNION

When a memory area is to be redefined with a variety of different representations, the use of a union greatly simplifies, not only the description, but also its ultimate replication Considering the following example of a hypothetical unification of all basic abal datatypes, leaving aside the sizes of the elements, a general approach could be declared using standard ABAL data declaration instructions as follows.

```
DCL BUFFER$
FIELD=M, BUFFER
   DCL ASCII$
FIELD=M, BUFFER
   DCL BCD=8
FIELD=M, BUFFER
   DCL INT8#
FIELD=M, BUFFER
   DCL INT16%
FIELD=M, BUFFER
   DCL INT32 :
FIELD=M, BUFFER
```

```
  DCL INT64&
FIELD=M
```

Naturally its replication would require:

```
DCL BUFFER2$
FIELD=M, BUFFER2
  DCL ASCII2$
FIELD=M, BUFFER
  DCL BCD2=8
FIELD=M, BUFFER
  DCL INT82#
FIELD=M, BUFFER
  DCL INT162%
FIELD=M, BUFFER
  DCL INT322 :
FIELD=M, BUFFER
  DCL INT642&
FIELD=M
```

The same results can be achieved by unification:

```
#UNION BUFFER
  DCL ASCII$
  DCL BCD=8
  DCL INT8#
  DCL INT16%
  DCL INT32 :
  DCL INT64&
#UNION BUFFER
BUFFER A
BUFFER B
BUFFER C
```

The advantages of the union, greatly simplifies the replication of the data types again alleviating all eventual adjacent anomalies.

The members of each structure being naturally accessible through the corresponding *“OBJECT NAME . MEMBER NAME”* construction, as shown below.

```
A.ASCII = “SOME TEXT”
B.BCD = 1.234
```

```
C.IN16 = 77
```

MIXED EXAMPLE

Unions and structures can be freely combined with basic data declarations withing the definitions of other unions and structures.

```
#STRUCT ANOTHER
ADDRESS A
BUFFER B
ADDRESS C
BUFFER D
ADDRESS E
#ENDSTRUCT ANOTHER
ANOTHER A
ANOTHER B
ANOTHER C
```

Notice that the names A, B and C, used within the structure definition are not in conflict with names of the instances of the structure ANOTHER. Access to individual member variables would be performed using the fully qualified object access path comprising “OBJECT NAME . MEMBER OBJECT NAME . MEMBER VARIABLE NAME” as shown below.

```
A.B.ASCII = “SOME TEXT”
B.D.BCD = 1.234
C.B.INT16 = C.D.INT16
A.A.TOWN = “PARIS”
B.C.TOWN = “NICE”
C.A.TOWN = A.E.TOWN
```

Please take the time you need, to feel comfortable with the above examples, before proceeding further. It is very important to fully understand the correspondence between the traditional manual declaration of structured data variables and the automation of these operations using the new UNION and STRUCT constructions.

CODE

Despite its early origins alongside similar primitive programming languages such as BASIC, B.A.L., and its subsequent successor ABAL, both offer a complete collection of instructions in support of STRUCTURED PROGRAMMING. It should be noted here, however, that the term STRUCTURED, in this sense, bears NO relation to the use of the term in the preceding section relation to the declaration of data and the new STRUCT and UNION constructions. The term STRUCTURED, here, refers simply to the careful imbrication, or nesting, of related structured programming instructions, such as IF/ELSE/ENDIF, WHILE/WEND, FOR/NEXT, SELECT/CASE, REPEAT/UNTIL, DO/LOOP, within a single logical section of code, as opposed to jumping around, back and forth, between close and distant regions of the program, using the much-abhorred, or much-revered, as your case may be, GOTO instruction.

STRUCTURED CODE EXAMPLE

The following section of code represents a hypothetical structured programming example:

```
DO
  WHILE E = 0
    FOR N = 1 to 10
      IF (( X = N ) OR ( E <> 0 ))
      ELSE
      ENDIF
    NEXT N
  WEND
LOOP
```

With each level of indentation used to represent the nested sub-structure of the code.

From the above example, and especially with the assistance of syntactical colouration, it can be clearly seen that the WHILE/WEND, FOR/NEXT and IF/ELSE/ENDIF statements, populating a particular SEGMENT or PROECDURE code region, are accessing variables and are consequently very intimately related with the actual structure of the accompanying data declarations. These variables may have been declared with either GLOBAL or LOCAL scope. These variables may also have been declared using the DATA declaration tools UNION and STRUCT as can be seen in the following section of code.

```
#STRUCT BUFFER
DCL E%
DCL N%
DCL X%
#ENDSTRUCT BUFFER
BUFFER A
DO
  WHILE A.E = 0
    FOR A.N = 1 to 10
      IF (( A.X = A.N ) OR ( A.E <> 0 ))
      ELSE
      ENDIF
    NEXT A.N
  WEND
LOOP
```

In which case it is immediately obvious that the subsequent section of code is intimately related with not only the definition of the structure, but also with the resulting instance of the structure itself, requiring complete replication of the code for application to an alternative instance even of the same structure. Naturally, the standard ABAL PROCEDURE would be an immediate choice for the

factorisation of this section of code, with redefinition of the received structured data parameter as a collection of traditional local variable declarations, as shown below.

```
PROC EXAMPLE(BUFFER$)
FIELD=M,BUFFER
DCL E%
DCL N%
DCL X%
FIELD=M
ENDLOC
DO
  WHILE E = 0
    FOR N = 1 to 10
      IF (( X = N ) OR ( E <> 0 ))
      ELSE
      ENDIF
    NEXT N
  WEND
LOOP
ENDPROC
```

Or could equally be described using a structure instance parameter.

```
PROC EXAMPLE (BUFFER P)
DO
  WHILE P.E = 0
    FOR P.N = 1 to 10
      IF (( P.X = P.N ) OR ( P.E <> 0 ))
      ELSE
      ENDIF
    NEXT P.N
  WEND
LOOP
ENDPROC
```

This is perfectly acceptable until the section of code would benefit from the ability to not only access different variables and values but also adapt itself to the actual very specific circumstances involved. To demonstrate this, consider the following example.

```
DCL BCD
```

```
DCL MSG$
PRINT=1:TAB(5,5),"VALUE:",BCD
PRINT=1:TAB(5,6),"MESSAGE:",MSG
```

This could not be so simply refactored as a procedure since the strict typed parameter, VAR$, of the procedure, would be refused the transfer of the BCD value, requiring a much more complex handling of the data or a replication of the procedure for the corresponding types.

```
DCL BCD
DCL MSG$
PROC DISPLAY(ATC%,ATL%,LABEL$,VAR$)
PRINT=1:TAB(ATCOL,ATLIN),LABEL,":",VAR
ENDPROC
DISPLAY(5,5,"VALUE",BCD)
DISPLAY(5,6,"MESSAGE",MSG)
```

This is naturally a very trivial example, but an added inconvenience would be incurred, in terms of performance since, the call to a procedure is slower than the execution of an INLINE section of code.

The next extension to ABAL, provided by ABAL++, in the form of the #MACRO/#ENDMACRO, was designed, especially, to alleviate this problem, with the ABAL compiler / translator, OTR, becoming responsible for replication of the section of code defined within the body of the macro, whilst performing substitution of the values of the parameters received by the invocation of the macro. The following example should clarify this.

```
#STRUCT BUFFER
DCL E%
DCL N%
DCL X%
#ENDSTRUCT BUFFER
#MACRO EXAMPLE(P)
DO
  WHILE P.E = 0
    FOR P.N = 1 to 10
      IF (( P.X = P.N ) OR ( P.E <> 0 ))
      ELSE
      ENDIF
    NEXT P.N
  WEND
LOOP
#ENDMACRO EXAMPLE
```

```
BUFFER A

BUFFER B

BUFFER C

EXAMPLE(A)

EXAMPLE(B)

EXAMPLE(C)
```

After careful examination of the preceding example, it should be clearly understood that the structure instances named A, B and C will be passed as the value of the parameter P, of the macro “EXAMPLE”, and will be substituted for all terms P within the code of the body of the macro. This will result in the replication of the structured programming instructions contained within the macro, each replication adapted to refer to the corresponding named structure, as determined by the value of the parameter P passed on invocation of the macro.

Please take the time you need, to feel comfortable with the above examples, before proceeding further. It is very important to fully understand the way in which the ABAL translator is capable of replicating sections of code while performing parameter pasting substitution.

This, again, is a very trivial example. Its use in actual programs is **not** encouraged and should **only** be used sparing, if at all, but its understanding is an important step in the journey, before moving on to the final section of this introduction to the fundamental requirements of object-oriented programming techniques.

CLASSES

The preceding sections relating to DATA and to CODE outline the transformations that were required to facilitate the replication of complex data structures and the integration of these data structures while replicating and adapting sections of code. Without an adequate organisational layer, applications, based on the data and code replication techniques described above, would still be operational but would present a rather ungainly mess of individual structure and macro definitions.

The final important extension to ABAL, provided by ABAL++, in the form of the #CLASS/#ENDCLASS, was designed, especially, to alleviate this problem by combining the powerful data and code replication techniques afforded by the STRUCT/UNION and MACRO definitions into a descriptive organisational unit known as a CLASS.

The following section of ABAL++ code shows a trivial example of a declaration of a class equivalent to the definition of a structure in the preceding DATA section.

```
#CLASS EXAMPLE

DCL INT8#

DCL INT16%

DCL INT32:

DCL INT64&

DCL BCD=8

DCL MSG$

PUBLIC INLINE DISPLAY:
```

```
  PRINT=1:INT8,INT16,INT32,INT64,BCD,MSG,TABV(1)
END
#ENDCLASS EXAMPLE
EXAMPLE E
E.DISPLAY
```

A CLASS definition may be composed of both data MEMBER definitions and code METHOD definitions and should be considered as being very similar to the definition of a new TYPE. Both MEMBER and METHOD definitions may be declared as PUBLIC, PRIVATE determining the scope of access allowed to the member from outside of the CLASS encapsulation, and COMMON determining the localisation of the actual member data variable or structure either inside the object container or as a shared common class variable.

MEMBER definitions may be described using either standard DCL type declaration syntax or extended STRUCT/UNION/CLASS syntax or any valid combination of the two.

METHOD definitions allow the organisation of blocks, or collections, of ABAL++ instructions into discrete functional units, of one of the five fundamental types INLINE, ROUTINE, FUNCTION, OVERLAY and USER FUNCTION, comprising a signature composed of its formal parameters and an eventual return type, and for all except for the case of the USER FUNCTION, the block of ABAL++ instructions delimited by the END keyword. In addition to the fundamental type, METHODS may be distinguished by their intended usage type CONSTRUCTOR, DESTRUCTOR, EXCEPTION, PRECONDITION, POSTCONDITION and INVARIANT. Methods may declare their nature of POLYMORPHIC ability or interaction using the STRICT and RELAX keywords. Finally, OVERLAY, FUNCTION and USER FUNCTION methods may be declared with INDIRECT, VIRTUAL, OVERLOAD or POINTER access attributes.

Classes are to be instanced in the same way as for structures and unions and their instance will be represented by an instance variable. A simple example of this can be seen below.

```
PROGRAM "EXAMPLE"
#CLASS EXAMPLE
DCL S$
PUBLIC INLINE METHOD HELLO: ( P$ )
  PRINT=1: ("Hello",$), P, TABV(1)
  s = p
END
#ENDCLASS EXAMPLE
EXAMPLE E
E.HELLO("world")
END
```

PUBLIC

CLASS members that are declared PUBLIC can be accessed from outside of the CLASS encapsulation and are said to contribute to the CLASS interface.

PRIVATE

CLASS members that are declared PRIVATE can only be accessed from inside the CLASS encapsulation, or from a CLASS that has been declared as a FRIEND, are said to contribute only to the CLASS implementation.

INHERIT

The INHERIT keyword allows an entire CLASS definition to be included, or absorbed, into the current CLASS definition. This keyword represents the concept of FUSIONAL inheritance. Subsequent CLASS MEMBER and METHOD definitions may accidentally or deliberately redefine the fundamental characteristics of members of the inherited class, transforming their behaviour accordingly. As a result of this operation, any of the FUNCTION and OVERLAY methods, originally defined in the inherited class, will become parented by the inheriting CLASS, giving rise to the generation of independent ABAL procedures and segments. The following shows an example of use of the INHERIT keyword.

```
#CLASS A
PUBLIC DCL V%
PUBLIC FUNCTION F:
END
#ENDCLASS A
#CLASS B
PUBLIC DCL FUNCTION MESSAGE$=256
INHERIT A
#ENDCLASS B
EXAMPLE A
EXAMPLE B
```

In the above example, objects of class A will comprise a single integer variable V and the corresponding procedure F, while objects of class B will comprise the integer variable V and the transformed procedure F.

PROTECT

This keyword may be associated with any MEMBER declaration statement and will signal that the corresponding member is to be protected against transformation, either deliberate or accidental, by an inheriting class. If the protected member has been defined in the inheriting class, then the member itself will be protected against transformation by any corresponding member inherited during subsequent INHERIT operations.

BASE CLASS

The alternative to FUSIONAL INHERITANCE is COMPOSITIONAL INHERITANCE more traditionally referred to as BASE CLASS inheritance. This is not represented by the keywords BASE and CLASS, instead it is described as shown in the following example.

```
#CLASS A
PUBLIC DCL V%
PUBLIC FUNCTION F:
```

```
END

#ENDCLASS A

#CLASS B (PUBLIC A)

PUBLIC DCL FUNCTION MESSAGE$=256

PUBLIC FUNCTION G:

END

#ENDCLASS B

EXAMPLE B
```

In this example, objects of class B will comprise a member object of class A containing single integer variable V and the corresponding procedure F, and a string variable and another procedure G. Further examples of the use of base class inheritance will be given in the description of the VIRTUAL and OVERLOAD methods later in this document. The PUBLIC keyword before the base class name indicates that the public members of the base instance are to be made visible for use as part of the derived classes public interface.

FRIEND

The FRIEND keyword allows a CLASS definition to declare the name of a friend class that will be allowed access to PRIVATE members as if it were a direct extension of the hosting class's own scope. This can be of importance when constructing complex base class hierarchies whilst ensure encapsulation of details of the actual implementation.

COMMON

The data storage elements of CLASS members that are declared COMMON will not be declared in the standard OBJECT container but will be declared uniquely in the GLOBAL variable space, to be shared between all objects of the same class, when declared PRIVATE COMMON, or by all objects of classes declaring the same COMMON member object, method, or variable, when declared PUBLIC COMMON.

The following section of ABAL++ code shows a trivial example of the use of the keyword COMMON.

```
 #CLASS EXAMPLE

PRIVATE DCL A$

PRIVATE COMMON DCL B$

PRIVATE DCL C$

#ENDCLASS EXAMPLE

EXAMPLE E

EXAMPLE F
```

The following section of ABAL code shows the declarations generated for the trivial example of the use of the keyword COMMON.

```
PROGRAM "EXAMPLE"

;** the common member variable

DCL        EB$

;** the instance E
```

```
PTR       E$=30
FIELD=M,E
          DCL EA$
          DCL EC$
FIELD=M
;** the instance F
PTR       F$=30
FIELD=M,F
          DCL FA$
          DCL FC$
FIELD=M
SEGMENT 0
;** construction of E
CREATE E
;** construction of F
CREATE F
ESEG 0
END
```

LIBRARY

The LIBRARY keyword allows the name of a dynamic library to be specified for the hosting class. This will identify the dynamic library destined to receive the collection of USER FUNCTION descriptions of the dynamic library functions.

METHOD

This optional keyword can be added to reinforce the fact that a METHOD declaration is a METHOD. The complete method declaration syntax, shown below, comprises the scope of the method as PUBLIC or PRIVATE, the POLYMORPHIC nature STRICT or RELAXED, the fundamental method type, as one of INLINE, ROUTINE, OVERLAY, FUNCTION or USER FUNCTION, the method access as METHOD, INDIRECT, POINTER, VIRTUAL, VIRTUAL or OVERLOAD, and the specific usage as CONSTRUCTOR, DESTRUCTOR, EXCEPTION, INVARIANT, PRECONDITION or POSTCONDITON.

```
; ** method declaration syntax components
{SCOPE} {POLYMORPHSM} {RETURNTYPE} {ACCESS} {TYPE} {USAGE} {name}: ({signature})
{INSTRUCTIONS}
END
```

The minimum declaration of a method requires a colon terminated naming token at very least. To resolve the ambiguity between a method and a 32bit integer variable, the variable must be declared using the DCL keyword. No white space is tolerated between the name terminating colon nor between the opening brace of the eventual signature parameters. This must be carefully respected because when space is present then the subsequent punctuation is presumed to be part of the method body

and not part of the signature definition. This distinction is of importance to allow the use of inline methods as token pasting macros where the macro text commences immediately after the method signature separated by a single space character without a preceding line feed. The signature, when present, will comprise a comma separated list of valid types, abal data types or class object types, between an opening and closing brace.

RETURN TYPE

A method return type, when present and allowed for INLINE, FUNCTION and USER FUNCTION methods, should be described using the CLASS {class name}, STRING, NUMERIC, INTEGER and POINTER keywords and should immediately follow any {SCOPE} and or {POLYMORPHISM} modifiers and before any {ACCESS} {TYPE} {USAGE} modifiers or the colon terminated naming token, as appropriate.

The following program source shows some simple examples of methods declaring return types.

```
#CLASS another
PUBLIC DCL buffer$
PRIVATE INLINE CONSTRUCTOR initialise:
    buffer = "buffer"
END
#ENDCLASS another
#CLASS EXAMPLE
PRIVATE DCL message$
PRIVATE DCL value%
PRIVATE another pointer P
PRIVATE INLINE CONSTRUCTOR initialise:
    message = "message"
    CREATE P
    value = 100
END
PUBLIC STRING FUNCTION get_message:
    EXIT (message)
END
PUBLIC INTEGER INLINE get_value:
    EXIT (value)
END
PUBLIC CLASS another pointer FUNCTION get_pointer:
    EXIT (P)
END
PUBLIC ROUTINE display:
    Print=1:message,value," : ",P.buffer,TABV(1)
```

```
END
#ENDCLASS EXAMPLE
PROGRAM "EXAMPLE"
EXAMPLE E
SEGMENT 0
    DCL name$
    DCL i%
    PTR p$
    ENDLOC
    name = E.get_message
    i = E.get_value
    p = E.get_pointer
    E.display
    Print=1:name,i," : ",p,tabv(1)
ESEG 0
END
```

Your attention is drawn to the use of the EXIT keyword to return a value from an INLINE METHOD and that an ALIAS is not required when returning a pointer from a METHOD declared as returning a pointer. The ALIAS will be handled implicitly by the ABAL translator.

INLINE

These methods are identical in operation to that of the MACRO. The result of their expansion will be generated into the currently active SEGMENT or PROCEDURE, with the automated substitution or resolution, during code expansion, of all references to data members of the same class, or other nested structures, and of all references to methods of the same class or nested structures. Inline methods may define formal parameters, which will be substituted during code expansion in the same way as for MACRO parameters.

Examples of INLINE methods can be found in the following sections describing the other keywords.

ROUTINE

These methods are identical in operation to that of the INLINE except that the result of the expansion, for a particular object instance, will be generated as a unique GOSUB/RETURN structure at the end of the hosting SEGMENT or PROCEDURE. A GOSUB instruction will be generated in the hosting SEGMENT or PROCEDURE at the position of invocation. These methods cannot define formal parameters nor return types. Automated substitution or resolution will be performed, during code expansion, of all references to data members of the same class, or other nested structures, and of all references to methods of the same class or nested structures.

Examples of ROUTINE methods can be found in the following sections describing the other keywords.

FUNCTION

These methods give rise to the generation of an ABAL PROCEDURE containing the code of the method. An implicit pointer variable will be defined as the first parameter of the PROCEDURE to receive a

pointer to the instance for which the FUNCTION method has been invoked. Automated substitution or resolution will be performed, during code expansion, of all references to data members of the same class, or other nested structures, and of all references to methods of the same class or nested structures. FUNCTION methods may define formal parameters, which will be substituted during code expansion in the same way as for MACRO parameters. FUNCTION methods may also define formal return values.

Examples of FUNCTION methods can be found in the following sections describing the other keywords.

OVERLAY

These methods give rise to the generation of an ABAL SEGMENT containing the code of the method. An implicit pointer variable will be defined as the first declaration of the LOCAL variable table and redefined with the appropriate class structure. A pointer to the invocation object will be passed to the OVERLAY method using a TCODE called register convention. These methods cannot define formal parameters nor return types. Automated substitution or resolution will be performed, during code expansion, of all references to data members of the same class, or other nested structures, and of all references to methods of the same class or nested structures.

USER FUNCTION

These methods represent dynamic library functions in the dynamic library defined by the string value of the LIBRARY instruction. The methods are expected to be in the library in the exact order of their declaration in the class. USER FUNCTION methods may define formal parameters and may also define formal return values. USER FUNCTION methods may not declare an END delimited collection of ABAL++ instructions.

CONSTRUCTOR

These methods will be invoked when an object of the hosting class is created, either:

- during the implicit construction of a natural CLASS instantiation statement (as shown in the preceding examples.)
- during the explicit construction of a CLASS object POINTER statement.

In both cases the constructor method, irrespective of its fundamental type, will be activated to ensure that the expected post-constructive state of the object is established.

Examples of constructor method declarations can be found in the example in the section relating to STRICT parameter checking.

DESTRUCTOR

These methods will be invoked when an object of the hosting class is destroyed, either:

- during the implicit destruction of a natural CLASS instantiation statement (as shown in the preceding examples.)
- during the explicit destruction of a CLASS object POINTER statement.

In both cases the destructor method, irrespective of its fundamental type, will be activated to ensure that the any residual state members are correctly released prior to the release of the object itself.

Examples of destructor method declarations can be found in the example in the section relating to STRICT parameter checking.

INDIRECT

When OVERLAY, FUNCTION or USER FUNCTION methods are declared INDIRECT, a corresponding ABAL SEGMENT, PROC or USER pointer variable will be declared and initialised, during the construction phase of the object life cycle, to point to the corresponding method. The resulting pointer variable will be used to allow indirect access to the method by all method invocation statements for corresponding method.

The following class object source:

```
#CLASS EXAMPLE

PUBLIC INDIRECT FUNCTION F:

END

#ENDCLASS EXAMPLE

EXAMPLE E
```

Would produce the following ABAL declarations:

```
PROGRAM "EXAMPLE"

PTR E$=9

FIELD=M, E

DCL PROC F

FIELD=M

PROC EXAMPLE_F(PTR E$=9) :: ENDPROC

SEGMENT 0

CREATE E

F = PROC PTR EXAMPLE_F

…
```

VIRTUAL

When OVERLAY, FUNCTION or USER FUNCTION methods are declared VIRTUAL, a method access vector comprising an object pointer and the corresponding ABAL SEGMENT, PROC or USER pointer variable will be declared and initialised, during the construction phase of the object life cycle, to point to the parenting object and the corresponding method. The resulting pointer variables will be used to allow indirect access to the method by all method invocation statements for corresponding method.

The following class object source:

```
#CLASS EXAMPLE

PUBLIC VIRTUAL FUNCTION F:

END

#ENDCLASS EXAMPLE

EXAMPLE E

E.F
```

Would produce the following ABAL declarations:

```
PROGRAM "EXAMPLE"

PTR E$=18

FIELD=M, E

DCL PROC EF

PTR EFP$=18

FIELD=M

PROC EXAMPLE_F(PTR E$=18) :: ENDPROC

SEGMENT 0

CREATE E

EFP = ALIAS(E)

EF = PROC PTR EXAMPLE_F

CALL (EF)(EFP)
```

OVERLOAD

When OVERLAY, FUNCTION or USER FUNCTION methods are declared OVERLOAD, and a corresponding VIRTUAL method is found to be accessible from the current base class hierarchy of identical method signature then the VIRTUAL method access vector will be overloaded, or redefined, and a new method access vector comprising the overloading object pointer and the corresponding ABAL SEGMENT, PROC or USER pointer variable will be declared and initialised, during the construction phase of the object life cycle, to point to the parenting object and the corresponding method. The resulting pointer variables will be used to allow overloaded indirect access to the method by all method invocation statements for corresponding method.

The following class object source:

```
#CLASS EXAMPLE

PUBLIC VIRTUAL FUNCTION F:

END

#ENDCLASS EXAMPLE

#CLASS DERIVED (EXAMPLE)

PUBLIC OVERLOAD FUNCTION F:

END

#ENDCLASS DERIVED

EXAMPLE D

D.F
```

Would produce the following ABAL declarations:

```
PROGRAM "EXAMPLE"

PTR D$=18
```

```
FIELD=M, D
DCL PROC EF
PTR EFP$=18
FIELD=M, EF
DCL PROC DF
PTR DFP$=18
PROC EXAMPLE_F(PTR E$=18) :: ENDPROC
PROC DERIVED_F(PTR E$=18) :: ENDPROC
SEGMENT 0
CREATE D
EFP = ALIAS(D)
EF = PROC PTR EXAMPLE_F
...
DFP = ALIAS(D)
DF = PROC PTR DERIVED_F
CALL (DF)(DFP)
```

The importance of this mechanism is not immediately visible in the trivial example above since the VIRTUAL/OVERLOAD mechanism is intended in support of much more complex situations requiring MULTIPLE INHERITANCE.

The following ABAL++ source example demonstrates the powerful advantages of the VIRTUAL/OVERLOAD construction in a trivial double base class derivation.

```
; ** definition of the base class ONE with two virtual methods
#CLASS ONE
PUBLIC FUNCTION CONSTRUCTOR I:
    PRINT=1:"CONSTRUCTION ONE",TABV(1)
    PRINT=1:"-ONE.I",TABV(1)
; ** the constructor I of class ONE invokes the methods A and B
    A()
    B()
END
PUBLIC VIRTUAL FUNCTION A:
    PRINT=1:"-ONE.A",TABV(1)
END
PUBLIC VIRTUAL FUNCTION B:
    PRINT=1:"-ONE.B",TABV(1)
```

```
; ** the method B invokes the method A
    A()
END
#ENDCLASS ONE
;** definition of the base class TWO with two virtual methods
#CLASS TWO
PUBLIC FUNCTION CONSTRUCTOR I:
    PRINT=1:"CONSTRUCTION TWO",TABV(1)
    PRINT=1:"-TWO.I",TABV(1)
; ** the constructor I of class TWO invokes the methods C and D
    C()
    D()
END
PUBLIC VIRTUAL FUNCTION C:
    PRINT=1:"-TWO.C",TABV(1)
; ** the method C invokes the method D
    D()
END
PUBLIC VIRTUAL FUNCTION D:
    PRINT=1:"-TWO.D",TABV(1)
END
#ENDCLASS TWO
; ** definition of the derived class DERIVED with base class inheritance of classes ONE and TWO
#CLASS DERIVED (PUBLIC ONE, PUBLIC TWO)
; ** overloading of the virtual method B of class ONE
PUBLIC OVERLOAD FUNCTION B:
    PRINT=1:"-DERIVED.B",TABV(1)
; ** the overloading method B invokes the method A of class ONE
    A()
;** and the method C of class TWO effectively, and formally,  bridging between the two class concepts
    C()
END
; ** overloading of the virtual method D of class TWO
PUBLIC OVERLOAD FUNCTION D:
    PRINT=1:"-DERIVED.D",TABV(1)
```

```
; ** the overloading method D invokes the original overloaded method D of class TWO
      TWO..D()
; ** note the overloaded invocation operator comprising two period characters ".."
; ** accessing the original member method of the class
END
#ENDCLASS DERIVED
PROGRAM "OVERLOAD"
DERIVED O
SEGMENT 0
PRINT=1:"SEGMENT 0",TABV(1)
PRINT=1:"CALL O.A",TABV(1)
O.A
PRINT=1:"CALL O.B",TABV(1)
O.B
PRINT=1:"CALL O.C",TABV(1)
O.C
PRINT=1:"CALL O.D",TABV(1)
O.D
PRINT=1:"ESEG 0",TABV(1)
ESEG 0
END
```

Would produce, excluding the PRINT statements for reasons of clarity, the following equivalent ABAL data declarations and code instructions. The redefinition of the object pointer parameters, received by each of the member procedures, has also been excluded for clarity. Their structure is identical to the appropriate global redefinition.

```
PROGRAM "EXAMPLE"
;** derived class instance
PTR O$=72
FIELD=M, O
DCL       ONE$=36
DCL       TWO$=36
;** class one method vectors
FIELD=M, ONE
DCL PROC FA
PTR PFA
DCL PROC FB
```

```
PTR PFB
;** class one method vectors
FIELD=M, TWO
DCL PROC FC
PTR PFC
DCL PROC FD
PTR PFD
;** derived class overloaded method vectors
FIELD=M, FB
DCL PROC OFB
PTR OPFB$=72
FIELD=M, FD
DCL PROC OFD
PTR OPFB$=72
FIELD=M
;** procedures for methods of CLASS ONE
PROC ONE_I(PTR O$=36) :: CALL (FA)(PFA) :: CALL (FB)(PFB) :: ENDPROC
PROC ONE_A(PTR O$=36) :: ENDPROC
PROC ONE_B(PTR O$=36) :: CALL (FA)(PFA) :: ENDPROC
;** procedures for methods of CLASS TWO
PROC TWO_I(PTR O$=36) :: CALL (FC)(PFC) :: CALL (FD)(PFD) :: ENDPROC
PROC TWO_C(PTR O$=36) :: CALL (FD)(PFD) :: ENDPROC
PROC TWO_D(PTR O$=36) :: ENDPROC
;** procedures for methods of CLASS DERIVED
; ** note the effective and formal bridging between the sub objects made possible with the multiple inheritance
PROC DERIVED_B(PTR O$=72) :: CALL (OFA)(OPFA) :: CALL (FC)(PFC) :: ENDPROC

; ** note the use of the direct procedure call with the nested member object here, resulting from the use of the
PROC DERIVED_D(PTR O$=72) :: CALL TWO_D (ONE) :: ENDPROC
; ** overloaded invocation operator, "..", instead of an indirection using the procedure and object pointers
SEGMENT 0
;** allocation of the object container
CREATE D
;** construction of CLASS ONE
PFA = ALIAS(ONE)
```

```
FA = PROC PTR ONE_A
PFB = ALIAS(ONE)
FB = PROC PTR ONE_B
CALL ONE_I(ONE)
;** construction of CLASS TWO
PFC = ALIAS(TWO)
FC = PROC PTR TWO _C
PFB = ALIAS(TWO)
FB = PROC PTR TWO _C
CALL TWO_I(TWO)
;** construction of CLASS DERIVED
POFB = ALIAS(O)
OFB = PROC PTR DERIVED_B
POFD = ALIAS(O)
OFD = PROC PTR DERIVED_D
;** invocation of the interface methods
CALL (FA)(PFA)
CALL (OFB)(POFB)
CALL (FC)(PFC)
CALL (OFD)(POFC)
ESEG 0
END
```

The execution of the above program, with the original PRINT statements included, would give the following screen output.

```
CONSTRUCTION ONE
-ONE.I
-ONE.A
-ONE.B
-ONE.A
CONSTRUCTION TWO
-TWO.I
-TWO.C
-TWO.D
-TWO.D
SEGMENT 0
CALL O.A
-ONE.A
```

```
CALL O.B
-DERIVED.B
-ONE.A
-TWO.C
-DERIVED.D
-TWO.D
CALL O.C
-TWO.C
-DERIVED.D
-TWO.D
CALL O.D
-DERIVED.D
-TWO.D
ESEG 0
```

In the preceding example it should be noted that the size and value of the object pointer component of the method vectors is initialised to point to its parenting object. In the case of the base component TWO this pointer would continue to point to the base member object TWO and not to the parenting object DERIVED until the construction of the overloaded method pointers of the DERIVED class effectively repoints the object pointer component of the method vectors of B and D to the parenting object DERIVED. The same object pointer component selection is performed for all virtual and overloaded method invocation instructions with the result of changing the parent object pointer depending on the level of derivation.

The **perfect** comprehension of the mechanisms involved, in this exhaustive example of multiple inheritance virtual method overloading, is **essential** for the successful use of these object-oriented techniques in a real-world application situation.

POINTER

When OVERLAY, FUNCTION or USER FUNCTION method statement includes the POINTER keyword, the corresponding INDIRECT, VIRTUAL or OVERLOAD construction will be generated without being attached to any ABAL SEGMENT, PROC or USER during construction phase of the object life cycle. The resulting pointer variable may be used to point to any signature compatible METHOD and to allow indirect access to the method by all pointer member invocation statements for corresponding member method pointer.

The following example demonstrates the declaration, initialisation, and use of METHOD POINTER members for the invocation of member methods.

```
#CLASS EXAMPLE

PUBLIC VIRTUAL FUNCTION POINTER DISPLAY(X%,Y%,MSG$)

PUBLIC VIRTUAL FUNCTION POINTER ANOTHER(X%,Y%,MSG$)

PRIVATE FUNCTION ONE:(X%,Y%,MSG$)

    PRINT=1:TAB(X,Y),MSG

END

PRIVATE FUNCTION TWO:(X%,Y%,MSG$)
```

```
    PRINT=1:MSG,TABV(1)
END
PUBLIC INLINE CONSTRUCTOR INITIALISE:
    CREATE DISPLAY(ONE)
    CREATE ANOTHER(TWO)
END
#ENDCLASS EXAMPLE
PROGRAM "POINTER"
EXAMPLE E
SEGMENT 0
E.DISPLAY (1,2,"HELLO")
E.ANOTHER (3,4,"BONJOUR")
ESEG 0
END
```

Translation and execution of the preceding example will produce the following screen output.

```
HELLO           BONJOUR
```

STRICT

The keyword STRICT, when associated with a METHOD definition activates the POLYMORPHIC nature of the METHOD identifier and allowing definition of multiple methods of the same name but with differing call and return signatures. The actual method will be selected during invocation statement as defined by the number and nature of the parameters provided in the statement.

The following example shows how this can be useful in providing multiple construction signatures allowing a variety of parameter combinations to be offered.

```
#CLASS EXAMPLE
PRIVATE DCL VALUE%
PRIVATE DCL MESSAGE$
PUBLIC STRICT INLINE CONSTRUCTOR INIT:(I%, S$)
    VALUE = I
    MESSAGE = S
END
PUBLIC STRICT INLINE CONSTRUCTOR INIT:(I%)
    INIT(I,"DEFAULT")
END
PUBLIC STRICT INLINE CONSTRUCTOR INIT:(S$)
```

```
    INIT(1,S)
END
PUBLIC STRICT INLINE CONSTRUCTOR INIT:
    INIT(1,"DEFAULT")
END
PUBLIC STRICT INLINE DESTRUCTOR TERMINATE:(I%, S$)
END
PUBLIC STRICT INLINE DESTRUCTOR TERMINATE:(I%)
END
PUBLIC STRICT INLINE DESTRUCTOR TERMINATE: :(S$)
END
PUBLIC STRICT INLINE DESTRUCTOR TERMINATE:
END
PUBLIC INLINE DISPLAY:
    PRINT=1:(E,X,$,/1),VALUE,MESSAGE
END
#ENDCLASS EXAMPLE
PROGRAM "STRICT"
EXAMPLE A
EXAMPLE B(5)
EXAMPLE C("STRING")
EXAMPLE D(7,"BOTH")
SEGMENT 0
A.DISPLAY
B.DISPLAY
C.DISPLAY
D.DISPLAY
ESEG 0
END
```

Execution, after translation, of the above program example would give the following screen output.

```
 1 DEFAULT
 5 DEFAULT
 1 STRING
 7 BOTH
 1 DEFAULT
```

In the above example of use of the STRICT keyword, it should be noticed that the complexity of the signature is to be declared in decreasing order. The signature with the most parameters should be declared first and the signature with the least, or none, should be declared last. This is very important for the correct operation of the polymorphic alternative function selection mechanisms.

Furthermore, when destruction is also described, a destructor of identical signature must be declared, corresponding to each of the polymorphic constructors, to ensure that the fault tolerance and exception handling mechanisms are perfectly symmetrical with respect to their construction and destruction phases.

RELAX

This optional keyword is the counterpart of the STRICT keyword and can be used to explicitly signal a non-polymorphic method that may not define alternative call and return signatures. This is the default nature of any method, with respect to polymorphism, when the STRICT keyword has NOT been specified.

COMETHODS

The following keywords are used to declare a special type of METHOD that is used in conjunction with another hosting METHOD of the same CLASS. Methods of this type can only be declared as of INLINE type, since they are to be used in conditional clauses, error trapping and event detection constructions before, after or around the instructions of the hosting method. The hosting method will reference the COMETHOD via the use of a LOCAL statement, as will be seen in the following examples of each individual COMETHOD type.

EXCEPTION

An EXCEPTION method allows ERROR processing to be named and organised and associated with all other method definitions through LOCAL EXCEPTION statements. When a LOCAL EXCEPTON statement is added to a METHOD body, the corresponding EXCEPTION method will be developed, around the collection of ABAL++ instructions of the hosting METHOD, it will provide a powerful and automated ERROR catching construction allowing fault tolerant operation.

The following example shows the ABAL++ class declaration and instantiation of an EXCEPTION method usage example.

```
#CLASS EXAMPLE

DCL V%(2)

PRIVATE EXCEPTION catcher:

  PRINT=1: (“an error occurred”,X,E,/1),catcher

  THROW catcher

END

PUBLIC CONSTRUCTOR FUNCTION initialisaton:

  LOCAL EXCEPTION catcher

  PRINT=1: “Force a variable bounds error”,TABV(1)

  V(0) = 0

END
```

```
#ENDCLASS EXAMPLE

PROGRAM “EXAMPLE”

EXAMPLE E

END
```

The following examples shows the standard ABAL instructions describing the way in which the ERROR trap, in the PROCEDURE generated for the CONSTRUCTOR FUNCTION, will be declared.

```
PROC EXAMPLE_INITIALIZATION(PTR O$)
  FIELD=M,O
  DCL V%(1)
  FIELD=M
   DCL CATCHER%
  ENDLOC
   ON LOCAL ERROR GOTO &CATCHER, CATCHER
   DO
      PRINT=1: “Force a variable bounds error”,TABV(1)
      V(0) = 0
      BREAK
&CATCHER
      PRINT=1: (“an error occurred”,X,E,/1),catcher
      ON LOCAL ERROR ABORT
      ERROR CATCHER
   LOOP
ENDPROC
```

INVARIANT

An INVARIANT method allows parallel condition processing to be named and organised and associated with any other method definitions through LOCAL INVARIANT statements. When a LOCAL INVARIANT statement is added to a METHOD body, the corresponding INVARIANT method will be prior to the collection of ABAL++ instructions of the hosting METHOD, but within the same contextual scope, and will provide a powerful and automated logical fault detection mechanism in addition to standard error processing.

The following program shows the trivial use of an invariant to detect and display even numbers inside a parent method containing a FOR/NEXT loop.

```
#CLASS EXAMPLE

PRIVATE DCL i%

PRIVATE INVARIANT even: (( i and 1 ) = 0 )

DO
```

```
    PRINT=1:i
END
PUBLIC OVERLAY METHOD operation:
    LOCAL INVARIANT even
    FOR I = 1 to 256
    NEXT I
END
#ENDCLASS EXAMPLE
PROGRAM "INVARIANT"
EXAMPLE E
SEGMENT 0
    E.operation
ESEG 0
END
```

The conditional expression must be a well-formed, left, and right brace encapsulated condition for the implicit ON EVENT instruction. It may start on a new line or on the same line as the method declaration, but in the latter case must be separated from the method name terminating colon by white space. The conditional expression must be followed, on its own new line, by the nature of the EVENT body, either DO, for multiple entry, or THEN for single entry invariants. Following the event body nature, will be the instructions of the event body, that are to be performed whenever the invariant evaluates to TRUE. The end of the INVARIANT method block signals the end of the INVARIANT and shall be terminated when required by an implicit END EVENT. The instructions of the body of the hosting method will then be processed and then the ensemble will be terminated by an EVENT OFF instruction prior to exit from the hosting method. In accordance with these rules, the preceding example of an INVARIANT method would be developed in the SEGMENT generated for the OVERLAY METHOD as shown below.

```
SEGMENT operation
  ON EVENT ((I AND 1) = 0)
  DO
    PRINT=1:i
  END EVENT
  FOR I = 1 to 256
  NEXT i
  EVENT OFF
  RET.SEG
ESEG operation
```

PRECONDITION

A PRECONDITION method allows entry condition processing to be named and organised and associated with any other method definitions through LOCAL PRECONDITION statements. When a LOCAL PRECONDITION statement is added to a METHOD body, the corresponding PRECONDITION method will be developed prior to the invocation of the hosting METHOD which will only be performed if the precondition is deemed to be true.

The following program demonstrates the trivial use of a precondition that will only allow display of a class member value if is found to be an even number.

```
#CLASS EXAMPLE

PRIVATE DCL I%

PRIVATE PRECONDITION even:

(( I and 1 ) = 0 )

END

PRIVATE ROUTINE METHOD display:

    LOCAL PRECONDITION even

    PRINT=1:i

END

PUBLIC OVERLAY METHOD operation:

    FOR I = 1 to 256

        display

    NEXT I

END

#ENDCLASS EXAMPLE

PROGRAM "INVARIANT"

EXAMPLE E

SEGMENT 0

    E.operation

ESEG 0

END
```

POSTCONDITION

A POSTCONDITION method allows exit condition processing to be named and organised and associated with any other method definitions through LOCAL POSTCONDITION statements. When a LOCAL POSTCONDITION statement is added to a METHOD body, the corresponding POSTCONDITION method will be developed after the return from the invocation of the hosting METHOD.

The following program demonstrates a trivial example using a POSTCONDITION to detect event values on return from its hosting method.

```
#CLASS EXAMPLE
```

```
PRIVATE DCL I%
PRIVATE POSTCONDITION even: (( I and 1 ) = 0 )
    PRINT=1:" even"
END
PRIVATE ROUTINE METHOD display:
    LOCAL POSTCONDITION even
    PRINT=1:i
END
PUBLIC OVERLAY METHOD operation:
    FOR I = 1 to 256
        display
    NEXT I
END
#ENDCLASS EXAMPLE
PROGRAM "INVARIANT"
EXAMPLE E
SEGMENT 0
    E.operation
ESEG 0
END
```

Environment Variables

OTR Pragmas

TOKENSIZE

This pragma indicates the maximum length of naming tokens for variables, constants, and procedures.

```
#pragma tokensize <integer expression>
```

Naming tokens that are longer than the specified length will be silently truncated.

KEYWORD

This pragma directs the translator to allow or ignore the customary nature of the indicated keyword allowing the keyword to be used as a naming token.

```
#pragma keyword <keyword token> [ ON | OFF ]
```

LOCAL_CONSTANT

This pragma directive instructs the translator to allow or inhibit the use of a local constant table for the storage of all implicit constants encountered during the translation of subsequent code blocks.

```
#pragma local_constant [ ON | OFF ]
```

ECHO_ON

This pragma directive activates subsequent source tracing as described by the integer expression.

```
#pragma echo_on <integer expression>
```

ECHO_OFF

This pragma directive inhibits subsequent source tracing.

```
#pragma echo_off
```

ECHO

This pragma directive activates or inhibits subsequent target tracing as described by the integer expression.

```
#pragma echo <integer expression>
```

HEAP

This pragma directive is identical to the traditional #HEAP directive and allows the size of the HEAP memory of the program to be defined.

```
#pragma heap <integer expression>
```

FILES

This pragma directive is identical to the traditional #FILES directive and allows the size of the FILE ASSING table of the program to be defined.

```
#pragma files <integer expression>
```

STACK

This pragma directive is identical to the traditional #STACK directive and allows the depth of the BYTE, WORD, and POINTER stacks of the program to be defined.

```
#pragma stack <integer expression>
```

MEM

This pragma directive is identical to the traditional #MEM directive and allows the size in bytes of the temporary expression storage buffer of the program to be defined.

```
#pragma mem <integer expression>
```

PAGENUMBER

This pragma directive sets the current listing page number to the value indicated by the integer expression.

```
#pragma pagenumber <integer expression>
```

APLUS

This pragma directive instructs the ABAL translator to activate or inhibit the implicit public or private nature of common statements.

```
#pragma aplus [  ON | OFF ]
```

KEYBOARD_FLUSH

This pragma directive instructs the translator to indicate that the keyboard buffer is to be flushed on program exit, or not.

```
#pragma keyboard_flush  [ ON | OFF ]
```

ENHANCED

This pragma directive instructs the translator concerning object translation enhancements.

```
#pragma enhanced <keyword> [ ON | OFF ]
```

The following enhancements are possible in this version of the OPENABAL Object Translator.

ERRORS

In normal execution conditions, Procedure and Segment code blocks will report any un-trapped errors through the standard system log in /var/log/syslog using the standard system **SYSLOG** relay to ensure that this is possible no matter which current user. This has replaced the previous use of the error logging file in the /tmp directory which caused core dumps and exceptions when unauthorised users attempt to write to the log file. When this enhancement is activated, a default **ON LOCAL ERROR X_CATCHER, &LABEL** error catching construction, with an associated **ON LOCAL ERROR ABORT** and **ERROR X_CATCHER** instructions, will be generated for all procedure and segment code blocks. This means that any errors encountered will be raised through the nested procedure and segment stack to be presented to an eventual nested error handler. If no nested error handler is encountered, then the error will be reported through the **SYSLOG** channel.

OPTIMISE

This pragma directive instructs the translator concerning which optimisations are to be performed or inhibited.

```
#pragma optimise <keyword> [ ON | OFF ]
```

SEMAPHORES

This pragma directive defines the number of ABAL semaphores to be declared for the program.

```
#pragma semaphores <integer expression>
```

SWAP_BUFFERS

This pragma directive defines the maximum number of SWAP buffers to be used by the translator during translation of the program.

```
#pragma swap_buffers <integer expression>
```

TRACE

This pragma directive instructs the ABAL translator to activate or inhibit subsequent translation tracing.

```
#pragma trace [  ON | OFF ]
```

ANNOUNCE

This pragma directive instructs the ABAL translator, to generate ABAL PAUSE instructions announcing the entry into the code region described by the list of code bock types.

```
#pragma announce [ [ constructor | destructor | exception | common | routine | function |
overlay | inline | ALL ]  [ ON | OFF ] , ]
```

The resulting expression may be inclusive or exclusive by first specifying ALL ON or ALL OFF and then activating or disactivating the code block types of interest or to be excluded.

IGNORE_CASE

The pragma determines if the translator is to differentiate between uppercase and lowercase tokens. By default, uppercase and lowercase tokens, are treated as identical.

```
#pragma ignore_case [  ON | OFF ]
```

LIST

This pragma directive instructs the ABAL translator to activate or inhibit subsequent translation listing.

```
#pragma list [  ON | OFF ]
```

PRIORITY

This pragma directive instructs the ABAL translator to activate or inhibit subsequent use of the translation priority option.

```
#pragma priority [  ON | OFF ]
```

WARNINGS

This pragma indicates which warnings are to be raised or inhibited by the translator when they are encountered.

```
#pragma warnings [ ALL ] [  ON | OFF ]  [ warning number, warning number ]
```

EDITOR

This pragma directive instructs the translator of the name of the text editor to be used when performing automated error tracing.

```
#pragma editor <filename>
```

ERRORS

This pragma directive instructs the translator to abandon translation if the upper error limit, indicated by the integer expression, here is reached.

```
#pragma errors <integer expression>
```

THROW

This pragma directive instructs the translator to use the value provided by the integer expression for implicit THROW instructions within the subsequent code blocks.

```
#pragma throw <integer expression>
```

SWAPSIZE

This pragma directive defines the maximum size of SWAP buffers to be used by the translator during translation of the program.

```
#pragma swapsize <integer expression>
```

PAGESIZE

This pragma directive sets the listing page size to the number of lines indicated by the integer expression.

```
#pragma pagesize <integer expression>
```

DEFINE

This pragma directive defines a program specific PRAGMA token to be accepted by the translator during subsequent source translation.

```
#pragma define keyword
```

UNDEF

This pragma directive cancels a previously defined program specific PRAGMA token. Subsequent use of the PRAGMA TOKEN expression will be signaled as an error.

```
#pragma undef keyword
```

OUTPUT

This pragma directive instructs the translator of the nature of the subsequent code to be output by the program production backed.

```
#pragma output [ C | CPLUS | JAVA | JPLUS ]
```

LABELSIZE

This pragma indicates the maximum length of label tokens.

```
#pragma labelsize <integer expression>
```

STYLE

Defines the global stylesheet to be used by the program by default.

```
#pragma style <filename>
```

INITLOCAL

Directs the translator to include or inhibit the generation of ABAL INITLOCAL instructions as the first instruction of a PROC or SEGMENT code block.

```
#pragma initlocal [  ON | OFF ]
```

NOFLUSH

This pragma directive instructs the translator to indicate that the standard one byte keyboard flush on program exit is to be respected, or not.

```
#pragma noflush  [ ON | OFF ]
```

NOINPUT

This pragma directive instructs the translator to inhibit the use of ASK, PAUSE and OP instructions to ensure that the PROGRAM cannot perform USER input.

```
#pragma noinput  [ ON | OFF ]
```

EXPORT

This pragma directive instructs the translator to export all subsequent procedure names such that they become visible for use by the dynamic library function call back mechanisms.

```
#pragma export  [ ON | OFF ]
```

CHARSET

This pragma directive instructs the translator to expect the specified character set to be used.

```
#pragma charset  [ ABAL | ANSI | UTF8 | UTF16 ]
```

DIFFERENCES

This section of the OPEN ABAL Language Reference outlines the known differences between ABAL64 and the previous architectural version ABAL32 describing the effects, the reasons, and how they are to be handled.

ABAL POINTER SIZE

In ABAL 2 (16bit) and ABAL 3 (32bit) the storage size of ABAL PTR variables was 5 bytes, comprising a leading POINTER TYPE byte and then the 32bit system pointer. In ABAL 64 the size of an ABAL PTR variable has been increased to 9 bytes, comprising a leading POINTER TYPE byte and then the 64bit system pointer. This decision was taken based on ensuring the maximum level of performance efficiency and robustness of the resulting ABAL Application. ABAL Sources are to be retranslated to ABAL64 programs for execution using the OPEN ABAL 64 Executer.

REGISTER INTEGERS

When integer values are loaded into a REGISTER of the ABAL virtual machine they become the same size of the ABAL PROGRAM word size. However, in previous versions this was then limited to 16 bits, for historic compatibility reasons, unless a call had been made to set **EVENT (998) to 1**. This has been inversed. A call to set **EVENT (998) to 0** is now required to force integers to 16 bits otherwise, in ABAL 64, they will be preserved as the natural integer size of the machine.

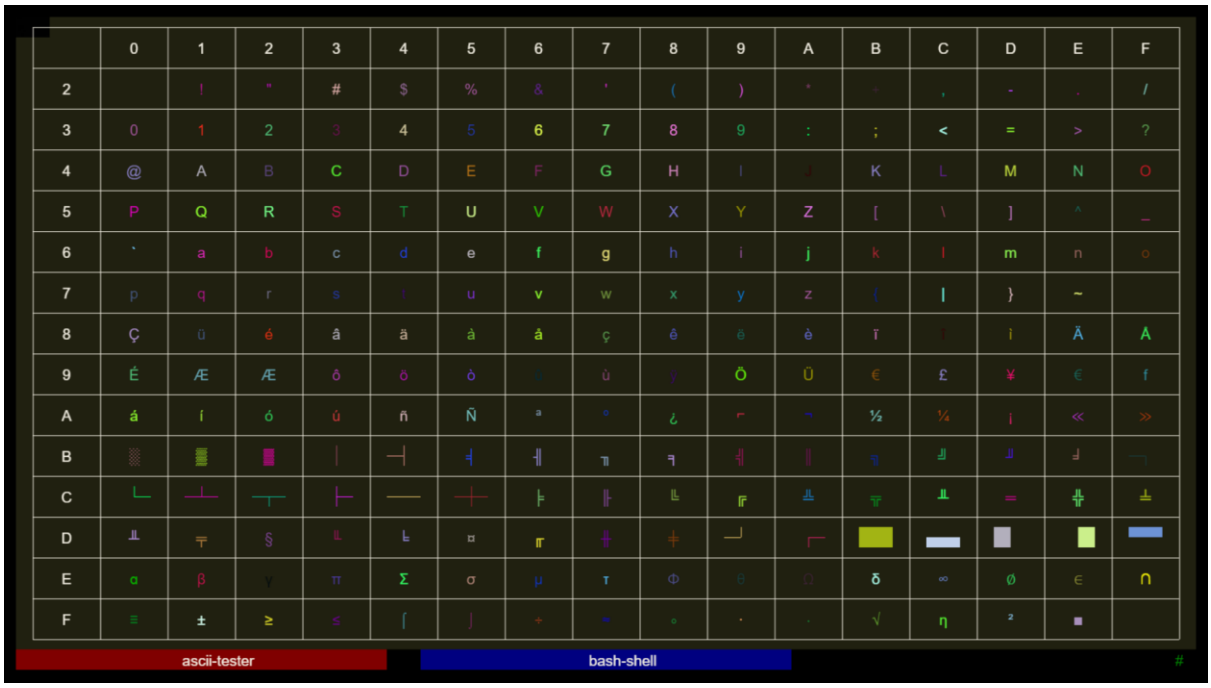
CLASS_NAME, OBJECT_NAME and METHOD_NAME

These pseudo functions return the string value of the corresponding CLASS, OBJECT or METHOD name or an empty string if not inside a CLASS, OBJECT or METHOD.

ANNEXE 1

The ABAL CHARACTER Set

The following table shows the standard ABAL character set.



ANNEXE 2

An XML File Copier

The following program, comprising the files xml.as and xml-example.as, demonstrates the use of the XML File Access instructions.

Xml.as

```
< constants for abal xml instruction >
Const XML_READ%=/0060
Const XML_READ_VERSION%=/0061
Const XML_READ_CHARSET%=/0062
Const XML_READ_STYLE%=/0063
Const XML_WRITE%=/00A0
Const XML_WRITE_VERSION%=/00A1
Const XML_WRITE_CHARSET%=/00A2
Const XML_WRITE_STYLE%=/00A3
Const XML_OPEN%=1
Const XML_NAME%=2
Const XML_VALUE%=3
Const XML_CLOSE%=4
Const XML_TEXT%=5
< eof >
```

Xml-example.as

```
program "xmlparser"
#include "xml.as"
dcl    e%
dcl    started%
dcl    input$=1024
dcl    output$=1024
dcl    version$=64
dcl    charset$=64
dcl    style$=1024
dcl    element$=256
dcl    buffer$=2048
field=m,buffer
      dcl    type%
      dcl    length%
      dcl    value$=2044
field=m
segment 0
      print=1:($,/1),"START ABAL XML PARSER TEST"
      input = "input.xml"
      Assign=1,input,XML:next,e
      Open=1:Next,e
      if ( e = 0 )
            print=1:($,/1),"OPEN XML : OK"
            output = "output.xml"
            Assign=2,output,XML,WR:next,e
            Cfile=2:Next,e
            if ( e = 0 )
                  print=1:($,/1),"CREATE XML : OK"
                  Read=1,XML_READ,0:Next,e,buffer,Len(buffer)
                  While ( e = 0 )
                        select ( type )
                        case    XML_OPEN
                              if ( started = 0 )
```

```
                        Read=1,XML_READ_VERSION,0:Next,e,version,len(version)
                        print=1:($,X,$,/1),"READ  XML VERSION = ",version
                        Write=2,XML_WRITE_VERSION,0:Next,e,version,len$(version)
                        Read=1,XML_READ_CHARSET,0:Next,e,charset,len(charset)
                        print=1:($,X,$,/1),"READ  XML CHARSET = ",charset
                        Write=2,XML_WRITE_CHARSET,0:Next,e,charset,len$(charset)
                        Read=1,XML_READ_STYLE,0:Next,e,style,len(style)
                        print=1:($,X,$,/1),"READ  XML STYLE   = ",style
                        Write=2,XML_WRITE_STYLE,0:Next,e,style,len$(style)
                        started = 1
                    endif
                    print=1:($,X,$,/1),"READ  XML : OPEN ELEMENT =",value
                    element = value
                    Write=2,XML_WRITE,0:Next,e,buffer,Len(buffer)
                case   XML_NAME
                    print=1:($,X,$,/1),"READ  XML : ATB NAME =",value
                    Write=2,XML_WRITE,0:Next,e,buffer,length+4
                case   XML_VALUE
                    print=1:($,X,$,/1),"READ  XML : ATB VALUE =",value
                    Write=2,XML_WRITE,0:Next,e,buffer,length+4
                case   XML_CLOSE
                    print=1:($,X,$,/1),"READ  XML : CLOSE ELEMENT =",value
                    Write=2,XML_WRITE,0:Next,e,buffer,length+4
                case   XML_TEXT
                    print=1:($,X,$,/1),"READ  XML : ELEMENT TEXT =",value
                    Write=2,XML_WRITE,0:Next,e,buffer,length+4
                endsel
                buffer = " " :: type = 0 :: length = 0
                Read=1,/60,0:Next,e,buffer,Len(buffer)
            wend
            Close=2:Next,e
            Close=1:Next,e
        else
            print=1:($,X,$,/1),"CREATE XML : FAIL",output
        endif
    else
        print=1:($,X,$,/1),"OPEN XML : FAIL ",input
    endif
    print=1:($,/1),"END ABAL XML PARSER TEST"
eseg 0
end
```

ANNEXE 3

A JSON File Copier

The following program, comprising the files json.as and json-example.as, demonstrates the use of the ABAL JSON Dynamic Library JSON File Access instructions.

Json.as

```
< JSON CONSTANTS for ABAL JSON LIBRARY >
const   JSON_NULL%=0
const   JSON_NAME%=1
const   JSON_VALUE%=2
const   JSON_STRUCT%=3
const   JSON_ARRAY%=4
const   JSON_CLOSE%=5
const   JSON_ERROR%=6
< EOF >
```

Json-example.as

```
#user "abaljson.def"
program "jsonparser"
#include "json.as"
dcl    type%
dcl    input%
dcl    output%
dcl    filename$=1024
dcl    newname$=1024
dcl    buffer$=2048
segment 0
      print=1:($,/1),"START ABAL JSON PARSER TEST"
      filename = "input.json"
      input = openjson(filename,len$(filename))
      newname = "output.json"
      output = createjson(newname,len$(newname))
      if ( input > 0 )
            print=1:($,/1),"OPEN  JSON : OK"
            if ( output > 0 )
                  print=1:($,/1),"CREATE JSON : OK"
            else
                  print=1:($,/1),"CREATE JSON : FAIL"
            endif
            while ( input > 0 )

                  type = readjson(input,buffer,len(buffer))
                  select ( type )
                  case    JSON_NULL
                        print=1:($,/1),"READ  JSON : NULL"
                        input = closejson(input)
                        output = closejson(output)
                        print=1:($,/1),"CLOSE JSON : OK"
                        input = 0
                  case    JSON_NAME
                        print=1:($,X,$,/1),"READ  JSON : NAME  =",buffer
                        type = writejson(output,type,buffer,len$(buffer));
                  case    JSON_VALUE
```

```
                    print=1:($,X,$,/1),"READ  JSON : VALUE =",buffer
                    type = writejson(output,type,buffer,len$(buffer));
                case   JSON_STRUCT
                    print=1:($,/1),"READ  JSON : STRUCT"
                    type = writejson(output,type,buffer,len$(buffer));
                case   JSON_ARRAY
                    print=1:($,/1),"READ  JSON : ARRAY"
                    type = writejson(output,type,buffer,len$(buffer));
                case   JSON_CLOSE
                    print=1:($,/1),"READ  JSON : CLOSE"
                    type = writejson(output,type,buffer,len$(buffer));
                case   JSON_ERROR
                    print=1:($,/1),"READ  JSON : ERROR"
                endsel
            wend
        else
            print=1:($,/1),"OPEN JSON : FAIL "
        endif
        print=1:($,/1),"END ABAL JSON PARSER TEST"
eseg 0
end
```

ANNEXE 4

A PARQUET File Example

The following snippet of ABAL source shows the values of the PARQUET data type constants.

```
< PARQUET CONSTANTS for ABAL PARQUET LIBRARY >
const   PARQUET_INT8%=1
const   PARQUET_INT16%=2
const   PARQUET_INT32%=3
const   PARQUET_INT64%=4
const   PARQUET_FLOAT%=5
const   PARQUET_DOUBLE%=6
Const   PARQUET_STRING%=7
< EOF >
```

The following ABAL sources shows a PARQUET file writer and reader example.

```
#user "abalparquet.def"
Program "AbalParquet"
#include "parquet.as"
Dcl     handle%
Dcl     err%
Dcl     filename$=1024
Dcl     buffer$=8192
Dcl     n%
Dcl     row&
Segment 0
        Print=1:Tab(1,1),"Abal Parquet Library Tester"
        filename = "/tmp/abal.parquet"
        Print=1:Tab(1,3),"Test of File Writer"
        ;** create the parquet file
        handle = ParquetCreate(filename,len$(filename))
        if ( handle > 0 )
                Print=1:Tab(1,4),("Parquet Create(",$,") : ",$),filename,conv$(handle)
                ;** create the file structure
                err = ParquetColumn(handle,parquet_int8,"byte",4,1,0)
                err = ParquetColumn(handle,parquet_int16,"word",4,2,0)
                err = ParquetColumn(handle,parquet_int32,"long",4,4,0)
                err = ParquetColumn(handle,parquet_int64,"huge",4,8,0)
                err = ParquetColumn(handle,parquet_float,"float",5,4,0)
                err = ParquetColumn(handle,parquet_double,"double",6,8,0)
                err = ParquetColumn(handle,parquet_string,"string",6,255,0)
                ;** write 1000 rows
                for row = 1 to 1000
                        n = row
                        buffer = Print(("(",$,",",$,",",$,",",$,",",$,",",$,",",$,",",$,",",$,",",$,",",$,",'A",HZ8,"')"),
conv$(n), conv$(n), conv$(n), conv$(n), conv$(n), conv$(n),n)
                        err = ParquetWrite(handle,buffer,len$(buffer))
                                Print=1:Tab(1,5),("Parquet
Write(",$,",",$,",$,")"),Conv$(row),buffer
                next row
                handle = ParquetFlush(handle)
                Print=1:Tab(1,6),("Parquet Flush (",$,") : ",$),filename,conv$(handle)
        Endif
        Print=1:Tab(1,8),"Test of File Reader"
        ;** read back the parquet data
        handle = ParquetOpen(filename,len$(filename))
        if ( handle > 0 )
                Print=1:Tab(1,9),("Parquet Open  (",$,") : ",$),filename,conv$(handle)
                Print=1:Tab(1,10),("Parquet Rows  (",$,") :
",$),filename,conv$(ParquetRows(handle))
                Print=1:Tab(1,11),("Parquet Cols  (",$,") :
",$),filename,conv$(ParquetColumns(handle))
                For row = 1 to ParquetRows(handle)
                        if ( ParquetRead(handle,row,buffer,len(buffer)) <> 0 )
                                Print=1:Tab(1,12),("Parquet Read
(",$,",",$,",$,")"),Conv$(row),buffer
                        else :: break
                        endif
                Next row
                handle = ParquetClose(handle)
                Print=1:Tab(1,13),("Parquet Close (",$,") : ",$),filename,conv$(handle)
        Endif
        Print=1:Tab(1,15),"Abal Parquet Library Tester"
Eseg 0
End
```