Technological University Dublin

ARROW@TU Dublin

Masters

Engineering

2001-09-01

Slab Drawing Layout: a Study to Develop a New Information Technology Package to Aid Those Involved in the the Manufacture of Pre-stresed Hollow Core Plank Flooring

Joe Parker

Technological University Dublin

Follow this and additional works at: https://arrow.tudublin.ie/engmas

Part of the Engineering Commons

Recommended Citation

Parker, J. (2001). *Slab drawing layout: a study to develop a new information technology package to aid those involved in the the manufacture of pre-stresed hollow core plank flooring.* Masters dissertation. Technological University Dublin. doi:10.21427/D79P65

This Theses, Masters is brought to you for free and open access by the Engineering at ARROW@TU Dublin. It has been accepted for inclusion in Masters by an authorized administrator of ARROW@TU Dublin. For more information, please contact arrow.admin@tudublin.ie, aisling.coyne@tudublin.ie, vera.kilshaw@tudublin.ie.

"SLAB DRAWING LAYOUT"

A study to develop a new Information Technology package to aid those involved in the manufacture of pre-stressed hollow core plank flooring.

Joe Parker B.Eng.

MPhil.

Dublin Institute of Technology

Mr. Eddie Fallon C.Eng., MSc.

Mr. Gerry Walker

School of Engineering Technology

September 2001

Volume 1 of 1

I certify that this thesis which I now submit for examination for the award of ____________________ , is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

This thesis was prepared according to the regulations for postgraduate study by research of the Dublin Institute of Technology and has not been submitted in whole or in part for an award in any other Institute or University.

The Institute has permission to keep, to lend or to copy this thesis in whole or in part, on condition that any such use of the material of the thesis be duly acknowledged.

Signature ____________________________________

Date ________________

ABSTRACT

Manufacture of pre-stressed hollow core flooring (from here on referred to as 'slabs') can be traced through a number of steps. These steps can generally be arranged in the following order;

1. Receipt of a drawing indicating the area that the client requires to be floored.
2. Production of a new drawing showing the layout of individual slabs over the proposed floor area.
3. Production of design calculations to B.S.8110 (or other design specification) that adequately covers each slab within the floor layout drawing according to its individual design criteria.
4. Production of sheets for manufacture of slabs showing the physical and geometrical properties of each individual slab.
5. Manufacture of slabs.
6. Transport of slabs from place of manufacture to site.
7. Placing of slabs on site.

Slabs are generally modular in nature and depending on the individual manufacturer's profile come in varying lengths of either 1200mm or 2400mm wide. The production of floor layout drawings thus tends to be repetitive which leaves them open for automation via computer technology. Since all floor slabs are of a uniform cross section their design will be dictated by their loading conditions and the presence or not of a structural screed. Design calculations can also be modeled using computer technology.

It was the intention of this study to automate as much as possible steps 2 to 4 from above thus reducing valuable time spent by manufacturer's staff on repetitive duties. In attempting this the study exposed the framework necessary to develop the IT package and steps 2 and 3 were adequately covered. An attempt was made to cover step 4 which worked for the simplest of cases. A very basic package was developed which could cater for uncomplicated small floor layouts. While most of the time was spent writing code to perform vital tasks the code needed to manipulate the user-interface was neglected. For industry standards this is a most

important area and a huge amount of thought would be necessary to develop this aspect of the package.

AutoCAD is the preferred drawing tool of the building industry and as such it was chosen as the main environment for the development of the Information Technology (referred to as IT subsequently) package. During the development of this thesis AutoCAD proved extremely versatile and the structure of its internal architecture made it a very powerful tool to work with.

To complement AutoCAD Borland's Delphi was chosen as the environment to write the non-drawing dependent software. The learning curve with Delphi is quite steep for a beginner but once mastered it has all the tools available for the most complex of software solutions.

Table of Contents

Chapter 1 Preliminaries ... *4*
1.1.Programming Environments ... 4
1.2.AutoCAD ... 4
1.3.So which API? ... 6
1.4.Borland Delphi ... 6
1.5.The Sample Project ... 7
Chapter 2: The First Approach ... *10*
2.1. The Problem At Hand ... 10
2.2. A Look At The Structure Of AutoCAD ... 11
2.3.Automation-The First Encounter ... 12
2.4. COM Technology – An Overview ... 13
2.5.AutoCAD and ActiveX Automation ... 13
2.6 About The Object Model ... 15
2.7. Working With Variants And Arrays ... 17
Chapter 3 Programming in VBA ... *19*
3.1. The VBA Environment ... 19
3.2 The Object Browser ... 21
3.3. Setting The Drawing Preferences ... 22
3.4 Adding Forms To The Project ... 23
3.5 System Variables ... 25
Chapter 4 Controlling Input ... *28*
4.1 Getting Input From The User ... 28
4.2 Error Trapping ... 29
4.3 Getting The Escape Key ... 30
4.4 Getting Keywords ... 33
4.5 Transparent Commands ... 38
Chapter 5 Programming Techniques ... *40*
5.1 Adding A Block To The Blocks Collection ... 40
5.2 Adding Objects To A Block ... 41
5.3 Connecting A Project To A Database ... 42
5.4 Implementing The Database With DAO ... 45
5.5 Automating Applications With AutoCAD ... 51
Chapter 6 The New Approach ... *54*
6.1 Dynamic Link Libraries – A Brief Introduction ... 54
6.2 Programmatically Creating A Toolbar ... 54
6.3 Adding A Macro To The Project ... 58
Chapter 7 The Toolbar ... *60*
7.1 A Description Of The New AutoCAD Toolbar ... 60
7.2 Get_Slab_Boundary ... 61
7.3 Get_Any_Holes ... 65
7.4 Get_Any_Walls ... 72
7.5 Get_Point_Loads ... 73
7.6 Get_Slabbed_Area ... 73
7.7 Hangers ... 75
7.8 Design ... 77
7.9 Group Slabs ... 79
7.10 Production Sheets ... 84
7.1 Toggle Layers ... 88
7.12 Reset ... 88
7.13 Concluding AutoCAD ... 88
Chapter 8 Delphi ... *89*
8.1 Where To Begin? ... 89
8.2 COM Revisited ... 89
8.3 The Basics Of COM ... 90

8.4 Virtual Method Tables 91
8.5 Interfaces 92
8.6 COM and the Client Server Model 95
8.7 Servers: In-Process and Out-of-Process 96
Chapter 9 Dynamic Link Libraries *97*
9.1 Type Libraries 97
9.2 Making A Start: The Simplest Of ActiveX DLLs 97
9.3 The Draw DLL 104
9.4 The Design Dll 109
Chapter 10 Conclusions *118*
10.1 Achieved Developments 118
10.2 Future Development 118
References *120*

Diagram Index

Fig 1.5.2 Isometric of 200mm deep unit 9
Fig 2.1.1 Inter-process Communication 10
Fig 2.2.1 AutoCAD' s database structure. 12
Fig 3.1.1 The Visual Basic Editor 19
Fig 3.2.1 The Object Browser 21
Fig 3.4.1 UserForm showing command button 24
Fig 3.4.2 The Properties Window 24
Fig 4.1.1 Floor Bounding Area 28
Fig 5.3.1 The ODBC Administrator 43
Fig 5.3.2 Setting up a Database 44
Fig 5.3.4 Available Project References 44
Fig 5.4.1 A Database Table 48
Fig 6.1.1 Available Macros Dialogue Box 58
Fig 6.2.1 Adding a Procedure 59
Fig 7.1.1 New Toolbar showing Tooltip for first button. 60
Fig 7.2.1 Client Name Dialogue Box 62
Fig 7.2.2 Floor Area Properties 63
Fig 7.3.1 Hole Sizing 65
Fig 7.3.2 Resultant hole added 67
Fig 7.3.3 Intersection area between floor area and hole 70
Fig 7.3.4 New hole produced 70
Fig 7.3.5 Floor area less new hole area 70
Fig 7.4.1 Wall Flowchart 72
Fig 7.5.1 Point load Flowchart 73
Fig 7.6.1 Slabbed floor area 75
Fig 7.7.1 Hangers Flowchart 76
Fig 7.9.1 Grouped Slabs 80
Fig 7.9.2 Group Slabs 83
Fig 10.2.1 Adding an ActiveX Library 98
Fig 10.2.2 Both member forms of the ActiveX dll 99
Fig 10.2.3 Automation Object Wizard 101
Fig 10.2.4 Delphi Type Library Editor 102

Fig 10.3.1 The Draw Editor 105
Fig 10.3.2 Dynamic Slicing 107
Fig 10.4.1 Results Sheet Toolbar 115
Fig 10.4.2 Changing the unit size 115
Fig 10.4.3 Changing the strand pattern 116
Fig 10.4.4 Creating cantilevers 116

Plate Index

Plate 7.10.1 Production Sheet 1 85
Plate 7.10.2 Production Sheet 2 86
Plate 10.4.2 Results Page No.1 112

Chapter 1 Preliminaries

1.1.Programming Environments

From the very outset of this research it was decided to work within two operating environments;

- AutoCAD
- Borland Delphi

1.2.AutoCAD

The reasons for choosing AutoCAD were straightforward.

1. AutoCAD is the universal drawing tool chosen by engineers worldwide. Most drawings received by a manufacture will either have been done using AutoCAD or can be interpreted using AutoCAD.
2. Over the years AutoDesk have been providing application-programming interfaces allowing users to get at the heart of its AutoCAD structure. It was known when this research began that AutoCAD could be programmed using one of these interfaces (or APIs for short) allowing the user to customise the programme for their own specific needs.
3. Two years ago when this research was started there was of a couple of APIs that could be used to programme AutoCAD. These were as follows;
    - Auto LISP,
    - ObjectARX AutoCAD Runtime Extension,
    - VBA and ActiveX Automation.

Auto LISP

AutoLISP is a programming language designed for extending and customizing AutoCAD functionality. It is based on the LISP programming language, whose origins date back to the late 1950s. LISP was originally designed for use in Artificial Intelligence (AI) applications, and is still the basis for many AI applications. AutoCAD introduced AutoLISP as an API in Release 2.1, in the mid-1980s. LISP was chosen as the initial AutoCAD API because it was uniquely suited for the

unstructured design process of AutoCAD projects, which involved repeatedly trying different solutions to design problems. (Note: Visual LISP has since been introduced which is a programming tool used to expedite AutoLISP programme development.)

ObjectARX or AutoCAD Runtime Extension

ObjectARX is the most powerful application development environment for AutoCAD. ARX was first introduced to AutoCAD in R13 and was much improved and enhanced for R14. This API is not for the beginner and requires the developer to have achieved a certain skill level. Knowledge of C++ including what virtual and overloaded functions are and how to derive a class from a base class is essential. The main benefit of ObjectARX is that custom AutoCAD objects can be created.

Microsoft Visual Basic for Applications or VBA

ActiveX Automation, a technology developed by Microsoft and based on the COM (component object model) architecture, is a new programming interface for AutoCAD. It can be used to customize AutoCAD, share drawing data with other applications, and automate tasks. Through Automation, AutoCAD exposes programmable objects, described by the AutoCAD Object Model that can be created, edited, and manipulated by other applications. Any application that can access the AutoCAD Object Model is an Automation controller, and the most common tool used for manipulating another application using Automation is Visual Basic for Applications (VBA). This form of Visual Basic is found as a component in many Microsoft Office applications. These applications, or other Automation controllers, such as Visual Basic and Delphi, can be used to drive AutoCAD. Microsoft VBA is an object-oriented programming environment designed to provide rich development capabilities similar to those of Visual Basic (VB). The main difference between VBA and VB is that VBA runs in the same process space as AutoCAD, providing an AutoCAD-intelligent and very fast programming environment. VBA was tentatively introduced to AutoCAD R14 it was not until

about two months after research began that full functionality was given to VBA in R14.01

1.3.So which API?

Research began by investigating the AutoLISP programming environment. About three weeks was spent examining sample AutoLISP applications and their associated code. The code is basically in the form of a long list delimited by commas and brackets written into a text editor and compiled by AutoCAD. It was found to be difficult to follow and extremely difficult to de-bug although 3 weeks would be considered a very short period in which to condemn it. At around the same time lots of references to VBA both within AutoCAD and also within many Microsoft products were beginning to appear. AutoCAD R14 that also had some sample VBA applications and associated code was available at this time. Although R14 did not have full VBA functionality it was decided that developing through it was the way forward. It has since been found that very little AutoLISP code was needed to complete the project. Due to the programming experience required using ObjectARX was never really considered. Charles McCauley published by AutoDesk Press for Thomson Learning extensively covers the topic in "Programming AutoCAD 2000 Using ObjectARX"

1.4.Borland Delphi

The reasons for choosing Delphi at the time were also quite straightforward;

- It was decided a more flexible programming environment was going to be used to perform all the calculations and to react to any user inputs with respect to automating the drawing process. Once all the calculations were done and an acceptable drawing layout was achieved the information would be sent back to AutoCAD for drawing.
- Due to the lack of programming experience advice as to what programming development tool to use was taken. Without any further research it was decided that Delphi would be the secondary development tool.

- Delphi is a rapid application development tool for the Windows environment. Delphi uses Object Pascal, a set of object-oriented extensions to standard Pascal. Object Pascal is a high-level (3-4GL), compiled, strongly typed language that supports structured and object-oriented design. Its benefits include easy-to-read code, quick compilation, and the use of multiple unit files for modular programming.
- The Visual Component Library (VCL) is a hierarchy of classes—written in Object Pascal and tied to the Delphi integrated development environment (IDE)—that allows applications to be developed quickly. Using Delphi's Component palette and Object Inspector, VCL components can be placed on forms and their properties manipulated without writing code.
- Most Delphi developers write and compile their code in Delphi's IDE. Delphi handles many details of setting up projects and source files, such as maintenance of dependency information among units. Delphi also places constraints on programme organisation that are not, strictly speaking, part of the Object Pascal language specification.

1.5.The Sample Project

Throughout the development of this study reference to the sample project is made. This basically refers to a section of a floor area that is required to be fitted with floor slabs by the manufacturer.

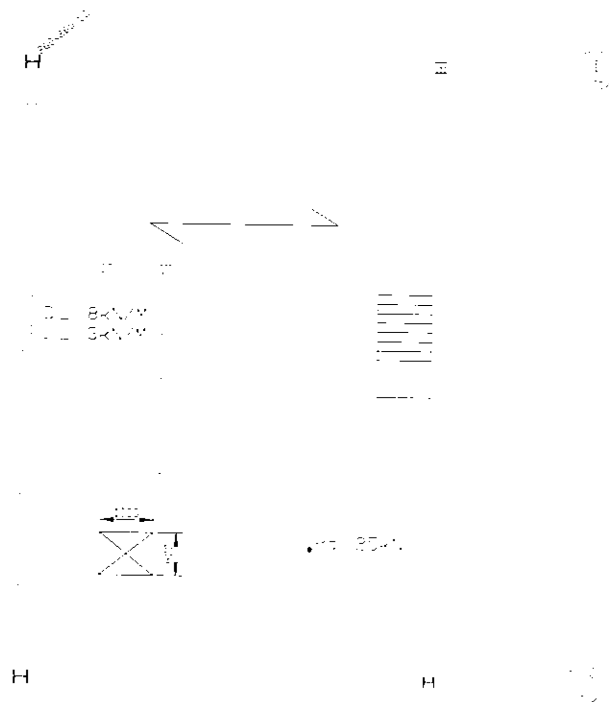


Fig 1.5.1 Demo.dwg

The drawing contains the following elements:

1. Gridlines B, C, X and Y.
2. A 260mm x 260mm Universal Column (UC) at gridline intersection points.
3. 2 No. Universal Beams between gridlines B and C.
4. Symbol indicating span of individual floor slabs.
5. 2 No. walls at right angles to each other imposing design loads of 8kN/m run Dead Load and 3kN/m run Live Load respectively.
6. A stairwell to another floor.
7. A large service opening measuring 1000 mm x 800 mm8. A point load imposing a design load of 25 KN Dead Load.

The development of this drawing into a group of intelligent "data aware" slab objects forms the basis of the research contained within this study. A slab object can be thought of as an individual slab record within the drawing containing the

entirety of design and geometric information needed by the manufacturer to reproduce a working unit on the shop floor.

Physically the floor slabs come as modular sections 1200mm wide of varying cross-sectional dimensions depending on the strength requirements of the floor area. The slabs contain varying numbers of prestressing strands used to increase the moment carrying capacity of the slabs. Once on site key joints at the edges of the slabs are filled with grout to tie adjacent slabs together.

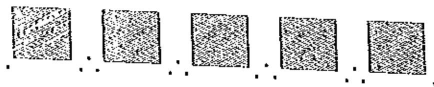


Fig 1.5.2 Isometric of 200mm deep unit

Chapter 2: The First Approach

2.1. The Problem At Hand

Looking at Demo.dwg the first challenge is obvious. The floor area contained within gridlines B, C, X and Y needs to be divided into a series of floor slabs of maximum manufacture width 1200mm and shown on the graphics screen. In order to achieve this it was decided that it was necessary to extract the coordinates of the outer extremities of the floor area and transfer them to the main programme where they would be manipulated and a revised set of coordinates sent back to AutoCAD for drawing.

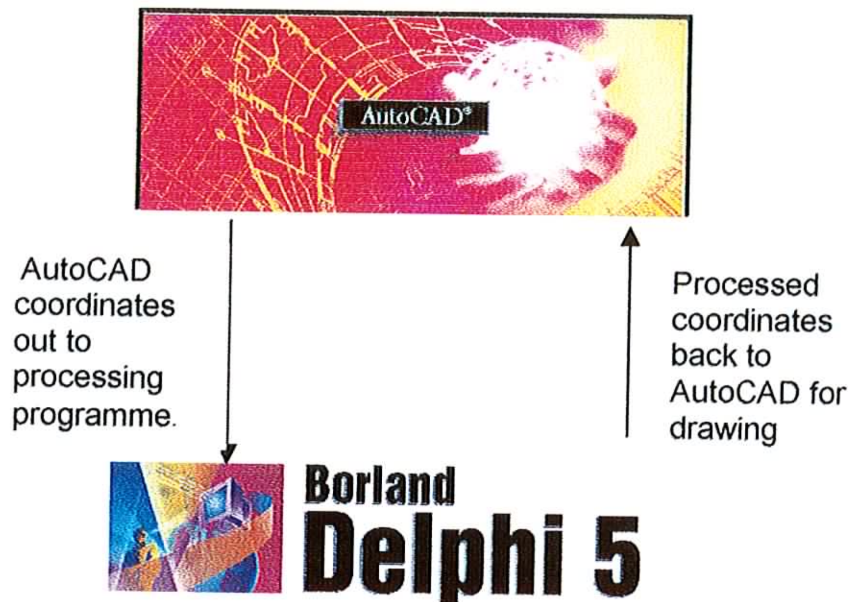


Fig 2.1.1 Inter-process Communication

The two programming environments used for the IT development. AutoCAD controls the drawing processes while Delphi manipulates the numbers before handing back to AutoCAD.

With this in mind it can be seen that development can assume a twin track approach with separate routines being developed for both the AutoCAD and Delphi sides of things.

2.2. A Look At The Structure Of AutoCAD

AutoCAD treats it's drawing more or less like a database. Within the drawing database there are a number of tables. Tables can contain records or index to other tables, which in turn have records. As with databases, the table can be opened for read or write operations. New records can be added (appended) to the table. Existing records can be modified (updated) in a particular table. When finished with whatever operation carried out on the table, the table must be closed. Thankfully VBA will take care of a lot of these operations behind the scenes but it is useful to know what is going on.

The AutoCAD drawing contains multiple objects some of which are visible to the graphics screen and some, which are not. Invisible objects are items such as layers and text styles. While a layer cannot be seen, its effects on the entities in the drawing residing on the particular layer can be seen. When starting a new AutoCAD drawing some of these objects are created automatically, such as the layer, the text style, the dimension style and so on. More importantly there is also a block table which has a few records already present: ***MODEL_SPACE** and ***PAPER_SPACE**. When the user executes an AutoCAD command to create an entity, AutoCAD creates an entity of the appropriate type, opens the correct table for a write operation and appends a new record to the table (the entity information is contained in the new record). Then AutoCAD closes the record and the database. The newly created entity will appear on the screen. AutoCAD does all of this in the background unknown to the user; it is hoped to demonstrate this using VBA.

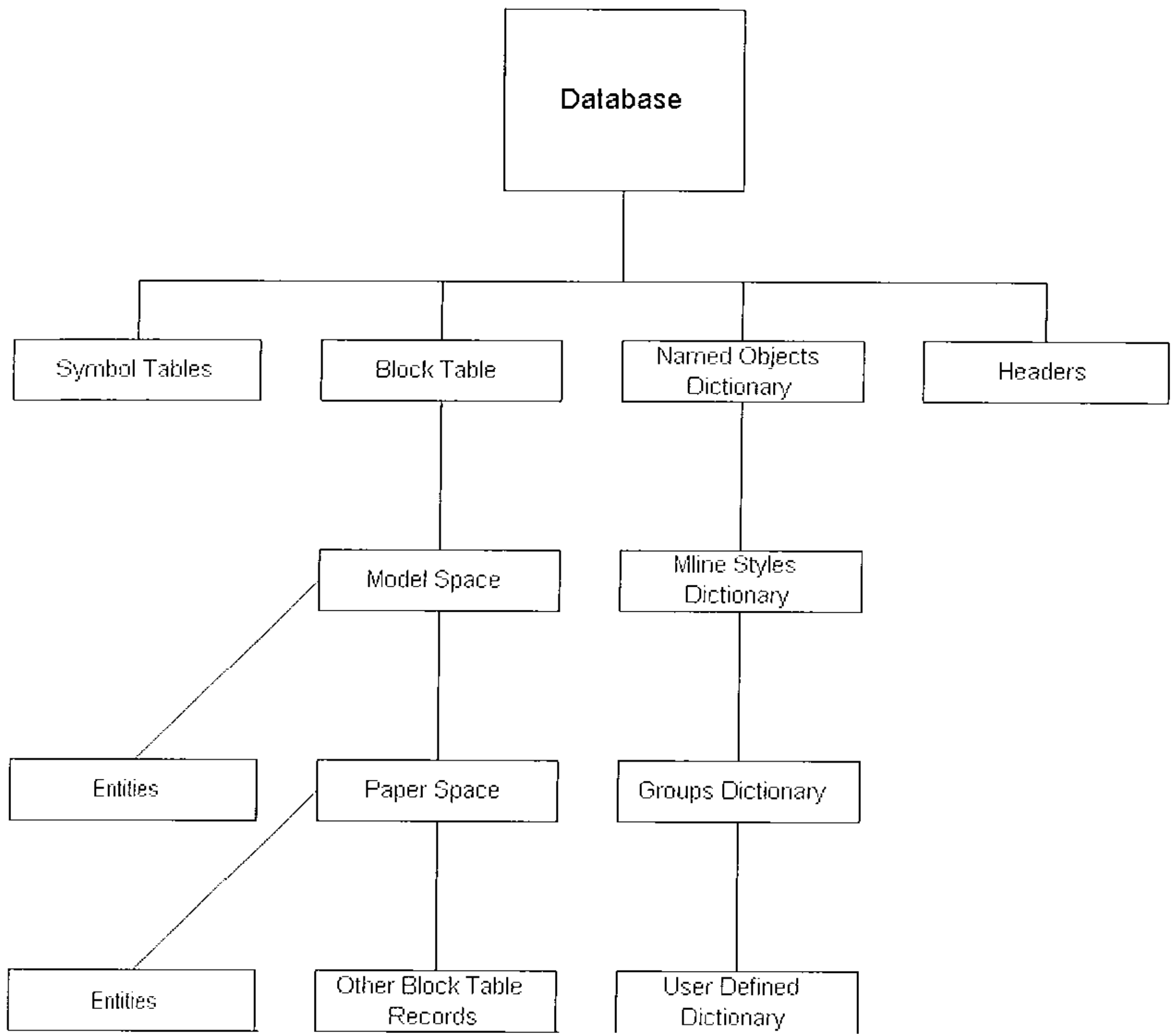


Fig 2.2.1 AutoCAD' s database structure.

Having briefly looked at the underlying structure of AutoCAD it is time to investigate some of the interfaces used to interact with this structure.

2.3.Automation-The First Encounter

AutoLISP was the first API considered to automate AutoCAD. About 3 weeks of development time was spent examining sample applications written in AutoLISP and developing applications. Two years ago AutoCAD R 14 was the platform on which development began. This was the first release of AutoCAD to include the

new Visual Basic API namely Visual Basic for Applications. Discovery of this interface put an end to investigations of programming AutoCAD using AutoLISP.

2.4. COM Technology – An Overview

COM stands for the Component Object Model. It is a system that was developed by Microsoft with the following theoretical purposes in mind:

- It provides a means for defining a specification for creating a set of non language-specific standard objects.
- It provides a means to implement objects that can be called between different processes, even if those processes are running on different machines.
- COM provides a platform across which different applications can communicate to each other. The full implications of COM to the development of this study will become apparent later on.

2.5.AutoCAD and ActiveX Automation

AutoCAD implements COM via its object model and ActiveX automation. ActiveX Automation provides a mechanism to manipulate AutoCAD programmatically from within or outside of AutoCAD. It does this by exposing various AutoCAD objects to the "outside world". Once these objects are exposed, they can be accessed by many different types of programming languages and environments, even by other applications such as Microsoft Word VBA or Excel VBA. VBA sends messages to AutoCAD by the AutoCAD ActiveX Automation Interface. AutoCAD VBA permits the VBA environment to run simultaneously with AutoCAD and provides programmatic control of AutoCAD through the ActiveX Automation Interface.

There are three fundamental elements that define VBA programming in AutoCAD. The first is AutoCAD itself, which has a rich set of objects that encapsulate AutoCAD entities, data, and commands. The second element is the AutoCAD ActiveX Automation Interface, which establishes messages (communication) with AutoCAD objects. The third element that defines VBA programming is VBA itself,

which has its own set of objects, keywords, constants, and so forth that provide program flow, control, debugging, and execution.

For simplicity the Object Model shown below is the one that was shipped with AutoCAD R 14.01.

Current development has taken advantage of the much-improved Object Model shipped with AutoCAD 2000.

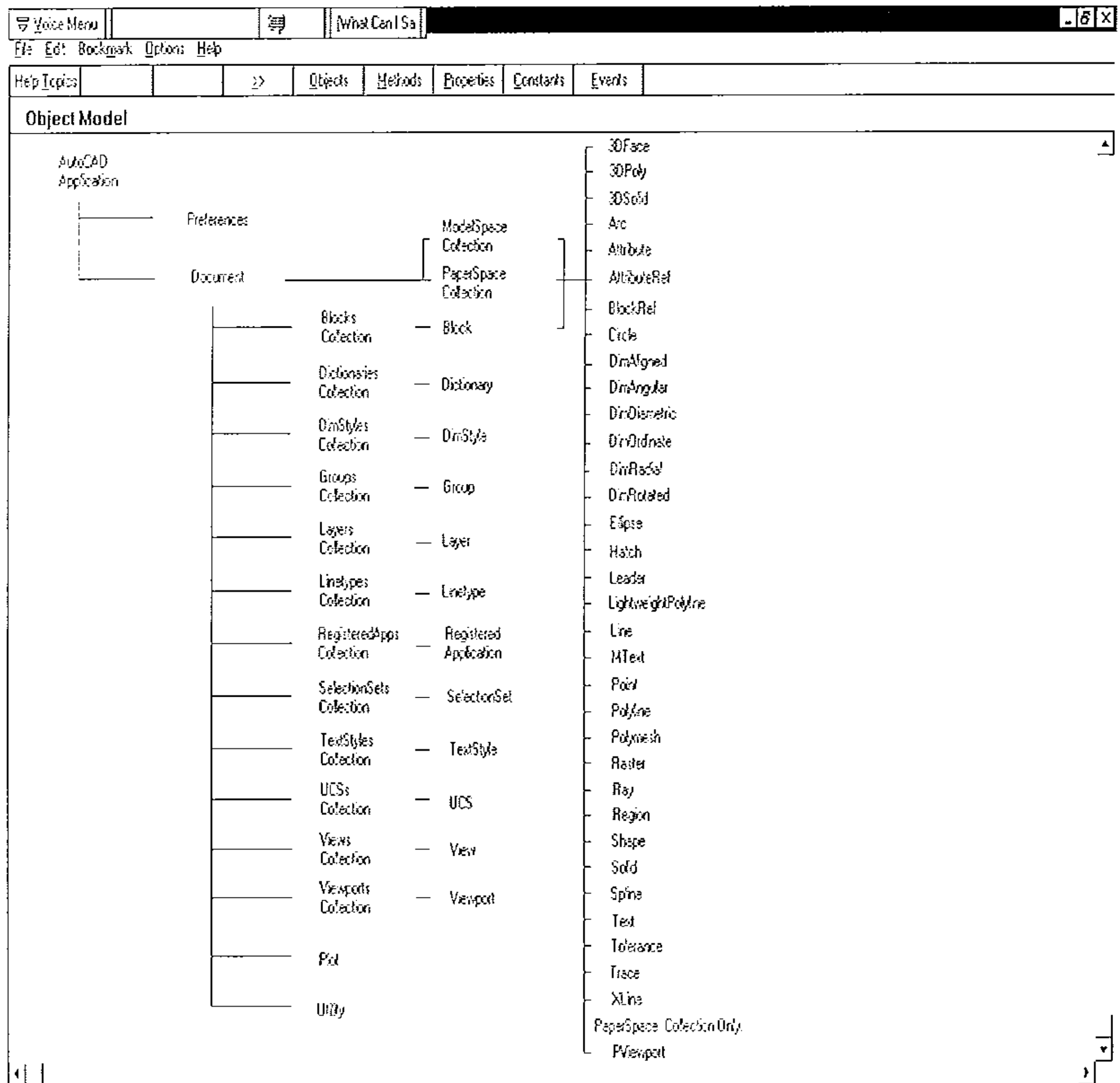


Fig 2.5.1 AutoCAD R 14 Object Model.[1]

2.6 About The Object Model

The Object Model is a hierarchical structure that provides programmable objects that can be accessed with various programming interfaces. An object can be thought of as one of the building blocks that go to make up the complete application.

Each object has its own individual properties and methods. Properties can be considered the attributes of an object whereas methods are usually applied to an object in order to modify it in some form or other. Each object has a parent object to which it is permanently linked. All objects originate from a single parent object called the root object. Following the links from the root to the child objects can access all the objects in the interface. Additionally, all objects have a property called Application that links directly back to the root object.

The root object for the AutoCAD interface is the AutoCAD application that is equivalent to an instance of the AutoCAD application running on the desktop. Consider AutoCAD with one drawing open (AutoCAD 2000 supports multiple drawings open at any one time) on the desktop. The appearance of the AutoCAD window as well as all the other non drawing-specific elements of the AutoCAD application can be accessed via the preferences object. The drawing object is equivalent to the .dwg file and contains the entire information within the drawing database. At any one time AutoCAD will have at least one drawing open that is considered the ActiveDocument of the application.

Every drawing object contains a group of collection objects that in turn contain the individual specification for each unique object within the collection. To illustrate this consider the blocks collection object. This collection will always have a minimum of two unique blocks within it namely the ***Model_Space** and the ***Paper_Space** blocks (although the names of these blocks will not show up if the user uses the InsertBlock toolbutton from AutoCAD). The most important of these is the ***Model_Space** block which contains the individual information for every object on the graphics screen.

Similarly the rest of the collection objects within the drawing may contain AutoCAD default objects. For example the DimStyles Collection will have at least one dimension style called "Standard", the layers collection will have at least one layer called "0" and so on. AutoCAD groups the graphical objects (lines, circles, arcs, and so forth) and non-graphical named objects (layers, linetypes, and so forth) in collections. Although these collections contain different types of data they can be processed using similar techniques. The **ModelSpace** and **PaperSpace** collections contain all of the graphical objects found in the drawing's model and paper space. The **ModelSpace** and **PaperSpace** collections have methods and properties for adding graphical entities, extracting a given item, and counting the number of objects in the collection. The non-graphical named objects are found in like-named collections, which also have methods and properties for adding, extracting, and counting items in each collection. The Blocks collection contains the list of all Block objects (block definitions or insert entities) in the drawing. Each Block object has methods and properties identical to those in the **ModelSpace** and **PaperSpace** collections, for maintaining the graphical objects in each block definition.

Let's say the user wants to draw a line in AutoCAD. The line toolbutton is chosen from the draw toolbar after which AutoCAD prompts to either choose coordinates for the start and end points on screen with the mouse or as data entered from the keyboard. Programmatically this could be duplicated with VBA code looking something like the following:

ThisDrawing.ModelSpace.AddLine (StartPoint, EndPoint)

Looking at this line of code an object called **ThisDrawing** can be seen which is at the top of the hierarchy – it owns a block called **ModelSpace** into which the user wants to add an AcadLine (the class to which all simple lines in AutoCAD belong) which has a defining StartPoint and EndPoint. StartPoint and EndPoint are actually variant variables containing the x, y and z coordinates. They are known as the parameters of the ***AddLine*** method or function.

In order to start programming in VBA (and when using Automation in general) an understanding of variants is essential.

2.7. Working With Variants And Arrays

A Variant is a special data type that can contain any kind of data except fixed-length string data and user-defined types. A variant can also contain the special values Empty, Error, Nothing, and NULL. How the data in a variant is treated can be determined using the VarType or TypeName Visual Basic functions. The Variant data type can be used in place of most any data type to work with data in a more flexible way. Variants are used to pass array data in and out of AutoCAD ActiveX Automation. This means that the arrays must be of the type Variant in order to be accepted by AutoCAD ActiveX Automation methods and properties. In addition, array data output from AutoCAD ActiveX Automation must be handled as a variant.

AutoCAD ActiveX Automation provides a utility method to convert an array of data into a variant. The method is called ***CreateTypedArray***. The ***CreateTypedArray*** method creates a variant that contains an array of integers, floating numbers, doubles, and so forth. The resulting variant can be passed into any AutoCAD method or property that accepts an array of numbers as a variant. Array information passed back from AutoCAD ActiveX Automation is passed back as a variant. If the data type of the array is known, tsimply access the variant as an array. If the data type contained in the variant is not known, use the Visual Basic functions VarType or Typename. These functions return the type of data in the variant. If it is necessary to iterate through the array, use the Visual Basic For Each statement. As well as these functions VBA provides the ***Lbound*** and ***Ubound*** functions that return the lower and upper bounds of the variant array respectively. By default arrays are zero indexed and thus the ***Lbound*** function returns zero for the array's lower bound. So a declaration of a variant array would look like this:

Dim myvar (0 To 5) As Variant

Here a variable called myvar is declared which is a one-dimensional array with six elements (0 indexed therefore element No. 1 = myvar (0)) each element of which is

a variant. In Visual Basic, arrays can be declared with up to 60 dimensions. For example, the following statement declares a 2-dimensional, 5-by-10 array.

Dim sngMulti(1 To 5, 1 To 10) As Single

The array can be thought of as a matrix, the first argument represents the rows and the second argument represents the columns.

In VBA dynamic arrays can be declared which don't get a dimension until runtime. A dynamic array is defined with open and closed parentheses remembering to give it a dimension at runtime.

Dim array () As Double

declares a dynamic array. To re-dimension an array use the Redim statement (also used to dimension the dynamic array) and if the integrity needs to be preserved of data already contained within an array to be re-dimensioned use the Preserve keyword.

Redim Preserve array (0 to 6) As Double

re-dimensions the existing array while preserving any data contained by any of the 7 individual array elements already existing.

Chapter 3 Programming in VBA

3.1. The VBA Environment

Unless otherwise stated it is assumed from here on that the user is working in an AutoCAD 2000 VBA environment. New VBA projects for AutoCAD are written within the Visual Basic Editor. This can be accessed by the following methods;

Tools menu: Macro-Visual Basic Editor
Command line: vbaide

AutoCAD displays the Visual Basic Editor.

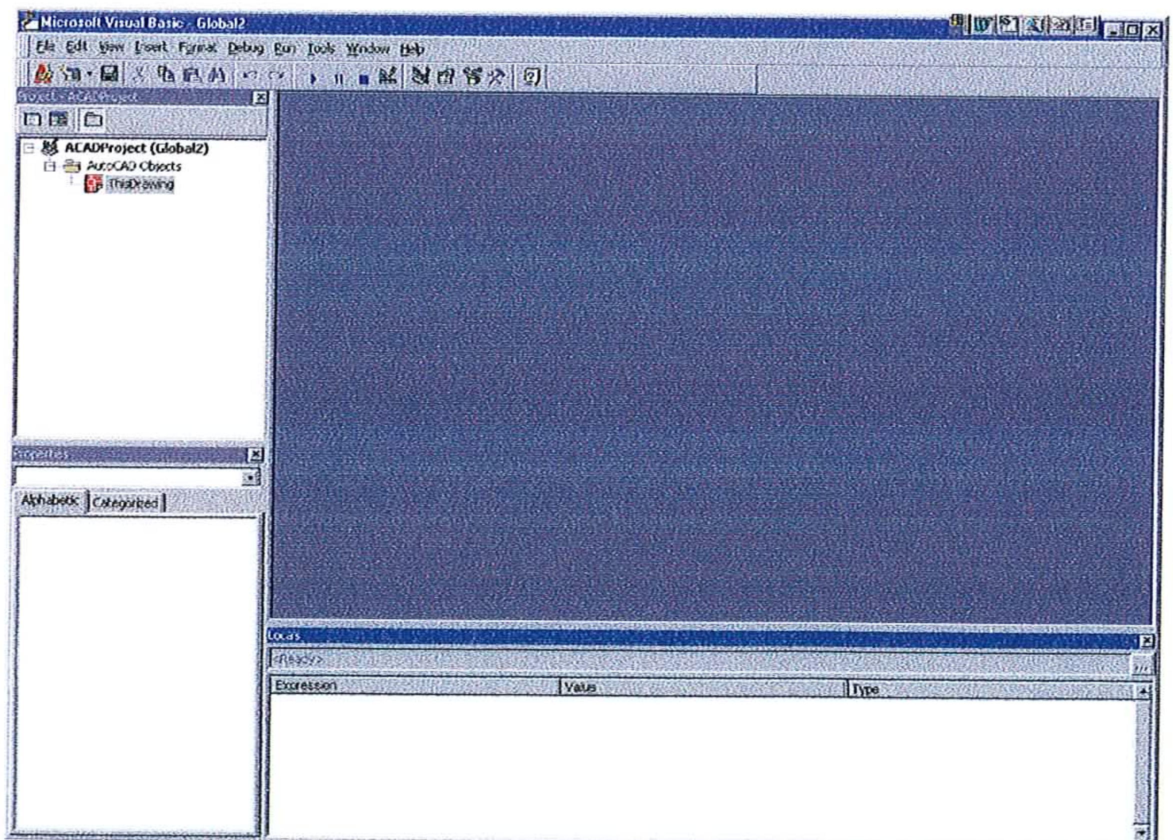


Fig 3.1.1 The Visual Basic Editor

The top left hand windowpane of the VBA window is known as the Project Explorer and is accessed via the View menu. The Project Explorer displays a hierarchical list of the projects and all of the items contained and referenced by each project. When a project is first started the explorer will show the name of the project and a folder called AutoCAD Objects. The only object contained within this folder at this time is the AutoCAD drawing which is open (and therefore the ActiveDocument) whose name defaults to ThisDrawing.

The most frequently used tools in this window are:

View Code

Displays the Code window so code can be written and edited associated with the selected item.

View Object

Displays the Object window for the selected item, an existing Document or UserForm.

Reverting to the sample project. The first approach to tackling the problem was to use database tables. The way in which these were to be implemented were as follows:

- The user would be asked to describe the perimeter of the floor area which was to be slabbed by placing the mouse cursor over a floor vertex and left clicking
- This floor area would then be converted to an AutoCAD polyline and added to the drawing's model space hence making the drawing aware of the floor area
- The coordinates of each vertex of the floor area would be sent to a database table which could be read by the Delphi programme
- Delphi would manipulate the coordinates producing new coordinates for the vertices of each individual slab and send these new coordinates back to the database table
- AutoCAD reads the coordinates from the new database table and draws the corresponding entities

3.2 The Object Browser

If not the most important tool used when programming in the VBA IDE, the most commonly used tool will be the Object Browser. This can be accessed via the tools menu:

View-Object Browser

Or by choosing the Object Browser tool button from the Standard toolbar. AutoCAD displays the Object Browser

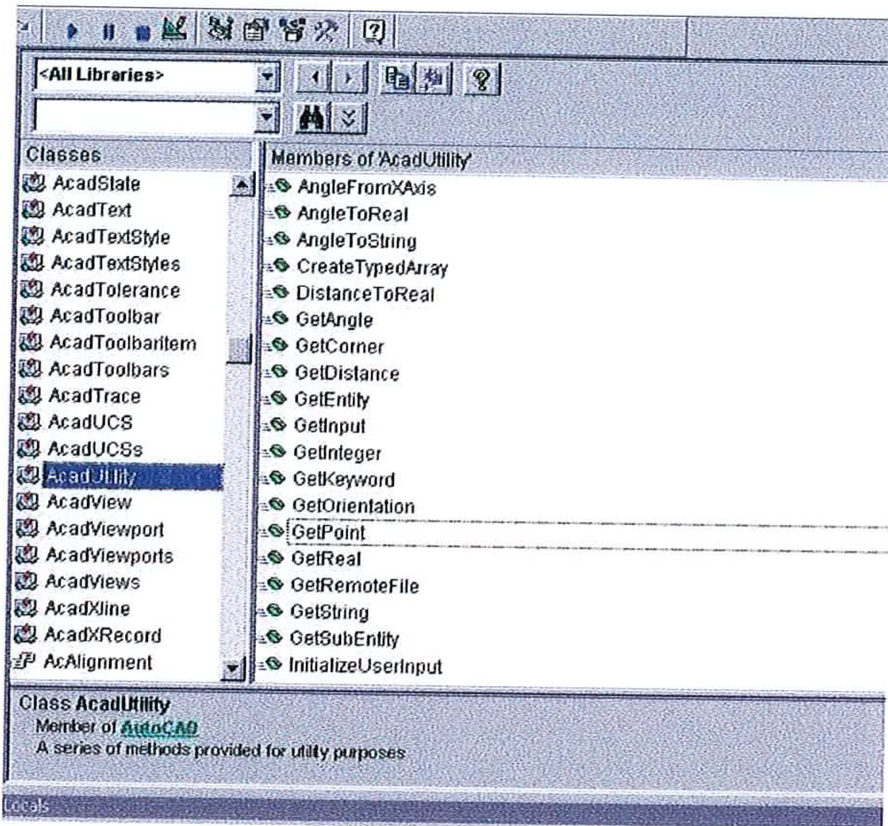


Fig 3.2.1 The Object Browser

The **"all libraries"** reference in the top left hand corner is a drop-down list that refers to all external libraries referenced by the current project. A library is basically a set of specifications for automatable objects that the project expects to use. The **Classes** window refers to all the available classes within the selected library. The class is the formal definition of an object. The class acts as the template from

which an instance of an object is created at run time. The class defines the properties of the object and the methods used to control the object's behavior. The **Members pane** displays the elements of the class selected in the **Classes** pane by group. Methods, properties, events, or constants that have code written for them appear bold. Right- clicking on any of the items in this pane will produce a drop-down list from which the associated help file can be accessed.

3.3. Setting The Drawing Preferences

The Preferences object holds all the options from the Options dialog that resides in the registry. Options that reside in the drawing can be found through the DatabasePreferences object. The Preferences object is divided into separate objects, with each representing a tab on the Options dialog. The Preferences object can be referenced from the Preferences property on the Application object:

```
Dim pref As AcadPreferences
Dim displaypref As AcadPreferencesDisplay
Set pref =ThisDrawing.Application.Preferences
Set displaypref = pref.Display
```

Of course a reference to the Display Preferences object can be obtained using the following:

```
Set displaypref = ThisDrawing.Application.preferences.DISPLAY
```

There are a couple of syntax conventions that should be noted at this time. When navigating through the Object Model a full stop or period (.) is used to separate the individual levels of hierarchy. Go to the tools menu and select Tools – Options will give a list of check boxes determining some environment options for the project. A useful box to have checked is the Auto list members box. When checked it automatically opens a drop-down list box in the Code window that contains the properties and methods available for the object

Finding the property or method wanted in the list box can be done by:

- Typing the name - while typing, the property or method that matches the characters typed is selected and moves to the top of the list.
- Using the up and down arrow keys to move up and down in the list.
- Scrolling through the list and selecting the property or method wanted.

The property or method can be inserted into the statement by:

- Double-clicking the property or method.
- Selecting the property or method and pressing TAB to insert the selection
- Pressing ENTER to insert the selection and move to the next line.

Note: Objects of the type Variant do not show a list after the period (.).

The Set keyword is always used when defining an object variable.

Drawing Preferences can be set according to the users choices, for example to change AutoCAD's default colours from white on black to black on white:

displaypref.GraphicsWinModelBackgrndColor = vbWhite [illegible]

To find out what the background colour is use the read version of the property:

Dim currGraphicsWinModelBackgrndColor As OLE_COLOR

currGraphicsWinModelBackgrndColor= displaypref.GraphicsWinModelBackgrndColor

More drawing preferences can be set similar to the above using the Object Browser.

3.4 Adding Forms To The Project

A form is an ActiveX control that can respond to events triggered by the user. Forms can also have other ActiveX controls placed onto them that in turn can be programmed to respond to input coming from a user.

A form can be inserted into a project from the tools menu:

Insert-UserForm

A blank UserForm appears in the right hand windowpane alongside which appears a toolbox with a list of available controls.

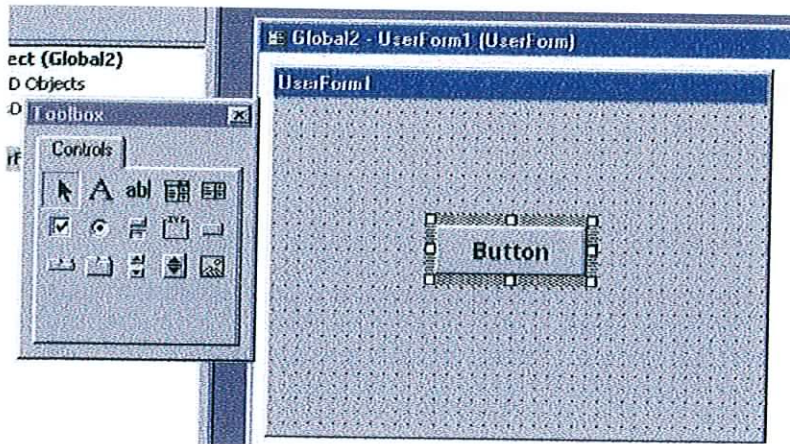
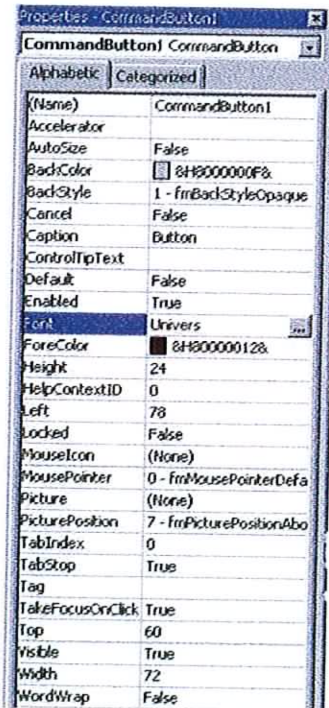


Fig 3.4.1 UserForm showing command button.

From the toolbox items can be dragged onto this UserForm that will respond to an action carried out by the user. In the UserForm above a command button has been dragged from the toolbox onto the center of the form and the caption "Button" has been placed onto the button. Both the UserForm and the command button (indeed all the ActiveX controls) have physical attributes that can be altered using the properties window. This is accessed via the tools menu **View-Properties Window.**



AutoCAD displays the properties window. Note how the caption property has been changed to read "Button" instead of the default "CommandButton 1". Similarly all the rest of the properties of the command button can be changed using the properties window. Clicking the drop-down list button will show all the other ActiveX controls in the project whose properties can be changed. Select a control in order to change its properties.

Fig 3.4.2 The Properties Window

So, in tackling the problem of getting the user to define the floor area while making the drawing aware of this area (by adding a polyline to its **ModelSpace**), the concept of UserForms was employed.

When a command button has been dragged onto the UserForm and then this button double-clicked a space will be given in which to write code. This code will then be executed when the user chooses to click on this button.

Private Sub CommandButton1_Click()

The code is written here to respond to the button click event.

End Sub

3.5 System Variables

The first step in processing the drawing from the sample project is to define the floor area to be covered with slabs by prompting the user to define its perimeter vertices. During drawing commands, the cursor can be snapped to points on objects such as endpoints, midpoints, centers, and intersections. For example, it is possible to turn on object snaps and quickly draw a line to the center of a circle, the midpoint of a polyline segment, or the apparent intersection of two lines. Since the user will be asked to define vertices it will be useful to have the ObjectSnap Mode turned on in the drawing. Exactly what snap settings are turned on depends on the value of the **OSMODE**[2] System Variable.

System variables control many AutoCAD functions and many aspects of the design environment. System variables control how many commands work. System variables also turn drawing modes (such as Snap and Grid) on or off, set default scales for hatch patterns, and store information about the current drawing and AutoCAD configuration. Each system variable has an associated type: integer, real, point, switch, or text string. For a full list of System Variables see the AutoCAD Command Reference.

The **OSMODE** System Variable is defined as follows:

Type: Integer

Saved in: Registry

Initial value: 4133

Sets running Object Snap modes using the following bitcodes.

0	NONe
1	ENDpoint
2	MIDpoint
4	CENter
8	NODe
16	QUAdrant
32	INTersection
64	INSertion
128	PERpendicular
256	TANgent
512	NEArest
1024	QUIck
2048	APParent Intersection
4096	EXTension
8192	PARallel

To specify more than one object snap, enter the sum of their values. For example, entering 3 specifies the Endpoint (bitcode 1) and Midpoint (bitcode 2) object snaps. Entering 16383 specifies all object snaps.

When object snaps are switched off using the Osnap button on the status bar, a bitcode of 16384 (0x4000) is returned, in addition to the normal value of OSMODE. With this additional value developers can write applications for AutoCAD, and distinguish this mode from Object Snap modes that have been turned off from within the Drafting Settings dialog box. Setting this bit toggles running object snaps off. Setting OSMODE to a value with this bit off toggles running object snaps on.

I turn on ENDpoint, MIDPoint, PERpindicular and APParent Intersection with the following code:

```
ThisDrawing.SetVariable "OSMODE", 2179
```

Where the ***SetVariable***[1] Method calls for two parameters:

object.***SetVariable*** Name, Value

Object Document

The object or objects this method applies to.

Name String; input-only

The name of the system variable to set.

Value Variant; input-only

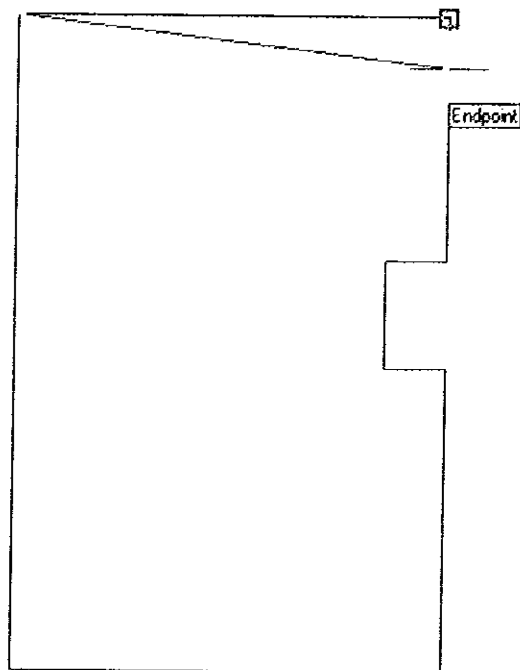
The new value for the specified system variable.

When setting system variables, AutoCAD may require integers, text, or double values. Passing the wrong data type, for example, passing a variant of type double when an integer is required, will also generate an error. The easiest way to avoid this is to use one of the Cxxx functions, such as **CInt()**, to explicitly type the data before it is passed.

Chapter 4 Controlling Input

4.1 Getting Input From The User

Once the desired environment options have been set then proceed by asking the user to define the floor area to be processed. AutoCAD provides a Utility Object that provides various methods for allowing the user to interact with the drawing. From the sample project it can be seen that the bounds of the floor area to the processed correspond to the following:



The Sample Project' s floor outline showing the red Endpoint Object Snap and the effect of rubber banding. There is obviously a command active in the drawing which is prompting the user to select a point on-screen while suggesting to select the highlighted Endpoint.

Fig 4.1.1 Floor Bounding Area

To get the coordinates of a point from the keyboard or the mouse use the Utility Function ***GetPoint*** that returns a 3 element variant array of type double.

Dim pnt As Variant pnt = ThisDrawing.Utility.GetPoint (prevpnt , "next point : ")

Where ***GetPoint***[1] calls for two parameters or arguments

pnt = object.GetPoint(prevpnt,[Prompt])

Object Utility

The object or objects this method applies to.

prevpnt Variant (three-element array of doubles); input-only; optional

The 3D WCS coordinates specifying the relative base point.

Prompt Variant (string); input-only; optional

The text used to prompt the user for input.

Pnt Variant (three-element array of doubles)

The 3D WCS coordinates of the point the AutoCAD user has selected.

AutoCAD pauses for user input of a point, and sets the return value to the value of the selected point. The prevpnt parameter specifies a relative base point in the World Coordinate System (WCS). The Prompt parameter specifies a string that AutoCAD displays before it pauses. Both prevpnt and Prompt are optional. The AutoCAD user can specify the point by entering a coordinate in the current units format; ***GetPoint*** treats the prevpnt parameter and the return value as three-dimensional points. The user can specify the point also by specifying a location on the graphics screen. If the prevpnt parameter is provided, AutoCAD draws a rubber-band line (A line that stretches dynamically on the screen with the movement of the cursor. One endpoint of the line is attached to a point in the drawing, and the other is attached to the moving cursor.) from prevpnt to the current crosshair position.

The coordinates of the point stored in the return value are expressed in terms of the WCS.

4.2 Error Trapping

When prompting the user for input consider what happens if the user supplies invalid input. This input can be dealt with using the concept of error handling. When using the ***GetPoint*** function AutoCAD expects an x, y, z coordinate from the keyboard or a point selected on the graphics screen using the digitiser. Any other input will be considered an error by AutoCAD and must be handled.

4.3 Getting The Escape Key

The first situation dealt with is if the user selects the Esc key. This key is the most commonly used key in AutoCAD to abort functions and as such its functionality should be reflected by any newly created toolbuttons. Invariably, depending on the situation in which the Esc key is selected, a different message will be sent back to AutoCAD. This makes it difficult to ascertain when the user has selected the Esc key and the way found to do this was to do the following:

- Declare a function at module level (describes code in the Declarations section of a module. Any code outside a procedure is referred to as module-level code. Declarations must be listed first, followed by procedures).
- Use a **On Error Resume Next** statement
- Test for the Esc key at runtime

```
Private Declare Function GetAsyncKeyState Lib "user32" (ByVal vKey As Long) As Integer
Private Const VK_ESCAPE = &H1B
On Error Resume Next
pnt =  ThisDrawing.Utility.GetPoint (prevpnt , "next point : ")
 If Err Then
  Err.Clear
   If GetAsyncKeyState(VK_ESCAPE) And &H8000& Then

  End If
End If
```

On Error Resume Next statements specify that when a run-time error occurs, control goes to the statement immediately following the statement where the error occurred where execution continues. This statement allows execution to continue despite a run-time error. Place the error-handling routine where the error would occur, rather than transferring control to another location within the procedure. An **On Error Resume Next** statement becomes inactive when another procedure is called, so it is necessary to execute an **On Error Resume Next** statement in each called routine to obtain inline error handling within that routine.

On Error GoTo Line and **On Error GoTo 0** are two other methods of trapping errors.

On Error GoTo line enables the error-handling routine that starts at line specified in the required line argument. The line argument is any line label or line number. If a run-time error occurs, control branches to line, making the error handler active. The specified line must be in the same procedure as the On Error statement; otherwise, a compile-time error occurs.

On Error GoTo 0 disables any enabled error handler in the current procedure.

Declare statements are used to access functions and procedures that are contained in external dynamic-link libraries (DLLs. A library of routines loaded and linked into applications at run time.).

They have the following form:

[Public|Private] ***Declare Function***[1] name Lib "libname" [Alias "aliasname"] [([arglist])] [As type]

Private Optional. Used to declare procedures that are available only within the module where the declaration is made.

Function Optional (either Sub or Function must appear). Indicates that the procedure returns a value that can be used in an expression.

name Required. Any valid procedure name. Note that DLL entry points are case sensitive.(***GetAsyncKeyState***[3])

Lib Required. Indicates that a DLL or code resource contains the procedure being declared. The Lib clause is required for all declarations.

libname Required. Name of the DLL or code resource that contains the declared procedure ("user32"- equates to c:\Windows\System\user32.dll on my system)

Alias Optional. Indicates that the procedure being called has another name in the DLL. This is useful when the external procedure name is the same as a keyword (a word with special meaning for the procedure as defined by the user). It is also possible to use Alias when a DLL procedure has the same name as a public variable, constant, or any other procedure in the same scope. Alias is also useful if any

characters in the DLL procedure name aren't allowed by the DLL naming convention.

aliasname	Optional. Name of the procedure in the DLL or code resource. If the first character is not a number sign (#), aliasname is the name of the procedure's entry point in the DLL. If (#) is the first character, all characters that follow must indicate the ordinal number of the procedure's entry point.
arglist	Optional. List of variables representing arguments that are passed to the procedure when it is called. (ByVal vKey As Long)
type	Optional. Data type of the value returned by a Function procedure; may be Byte, Boolean, Integer, Long, Currency, Single, Double, Date, String (variable length only), or Variant, a user-defined type, or an object type. (As Integer)

The arglist argument has the following syntax and parts:

[Optional] [ByVal I ByRef] [ParamArray] varname[( )] [As type]

Part	**Description**
Optional	Optional. Indicates that an argument is not required. If used, all subsequent arguments in arglist must also be optional and declared using the Optional keyword. Optional can't be used for any argument if ParamArray is used.
ByVal	Optional. Indicates that the argument is passed by value.
ByRef	A way of passing the address of an argument to a procedure instead of passing the value. This allows the procedure to access the actual variable. As a result, the variable's actual value can be changed by the procedure to which it is passed. Unless otherwise specified, arguments are passed by reference.
ParamArray	Optional. Used only as the last argument in arglist to indicate that the final argument is an Optional array of Variant elements. The ParamArray keyword allows an arbitrary number of arguments. The ParamArray keyword can't be used with ByVal, ByRef, or Optional.

varname	Required. Name of the variable representing the argument being passed to the procedure; follows standard variable naming conventions.(vkey)
( )	Required for array variables. Indicates that varname is an array.
type	Optional. Data type of the argument passed to the procedure; may be Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (not currently supported), Date, String (variable length only), Object, Variant, a user-defined type, or an object type.(As Long)

The ***GetAsyncKeyState*** function determines whether a key is up or down at the time the function is called, and whether the key was pressed after a previous call to ***GetAsyncKeyState***. It is provided by the Windows Advanced Programming Interface (API). It has the form:

SHORT	***GetAsyncKeyState***(vKey)
VKey	Specifies one of 256 possible virtual-key codes.
Rtn Values	If the function succeeds, the return value specifies whether the key was pressed since the last call to ***GetAsyncKeyState***, and whether the key is currently up or down. If the most significant bit is set, the key is down, and if the least significant bit is set, the key was pressed after the previous call to ***GetAsyncKeyState***. The return value is zero if a window in another thread or process currently has the keyboard focus.

4.4 Getting Keywords

Keywords are words which when typed at the Command Line in AutoCAD perform specific functions. VBA gives the programmer a chance to define their own keywords within their projects. Keywords are defined using the ***InitializeUserInput***[1] procedure again provided by the AutoCAD Utility Object.You can call the procedure with code similar to the following:

```
Dim keywd As Variant
Keywd = Array("case1","case2"......etc)
ThisDrawing.Utility.InitializeUserInput 1,keywd
```

The method has the following form:

object.***InitializeUserInput*** Bits[, Keyword]

Object Utility

The object or objects this method applies to.

Bits Integer; input-only

To set more than one condition at a time, add the values together in any combination. If this value is not included or is set to 0, none of the control conditions apply.

1: Disallows NULL input.

This prevents the user from responding to the request by entering only [Return] or a space.

2: Disallows input of zero (0).

This prevents the user from responding to the request by entering 0.

4: Disallows negative values.

This prevents the user from responding to the request by entering a negative value.

8: Does not check drawing limits, even if the LIMCHECK system variable is on.

This enables the user to enter a point outside the current drawing limits. This condition applies to the next user-input function even if the AutoCAD LIMCHECK system variable is currently set.

32: Uses dashed lines when drawing rubber-band lines or boxes.

This causes the rubber-band line or box that AutoCAD displays to be dashed instead of solid, for those methods that let the user specify a point by selecting a location on the graphics screen. (Some display drivers use a distinctive color instead of dashed lines.) If the POPUPS system variable is 0, AutoCAD ignores this bit.

64: Ignores Z coordinate of 3D points (GetDistance method only).

This option ignores the Z coordinate of 3D points returned by the GetDistance method, so an application can ensure this function returns a 2D distance.

128: Allows arbitrary input—whatever the user types.

Keyword Variant (array of strings); input-only; optional

The keywords that the following user-input method would recognize. Keywords must be defined with this method before the call to ***GetKeyword*** [1]. Certain user-input methods can accept keyword values in addition to the values they normally return, provided that this method has been called to define the keyword. The user-input methods that can accept keywords are: ***GetKeyword, GetInteger, GetReal, GetDistance, GetAngle, GetOrientation, GetPoint, and GetCorner***. There are two different ways of getting keywords from the user:

- Directly using the ***GetKeyword*** procedure provided by the Utility Object
- By Error Trapping

Use the ***GetKeyword*** procedure to prompt the user to input none other than a keyword from the keyboard:

rtnval = ThisDrawing.Utility.GetKeyword (prmpt)

prmpt Variant (string); input-only; optional

The text used to prompt the user for input.

Rtnval String

The keyword returned from the user.

AutoCAD pauses for user input of a **keyword** and sets the return value to the **keyword** that was entered. The Prompt parameter specifies a string that AutoCAD displays before it pauses. The prompt is optional. The maximum length of the return value is 511 characters. The AutoCAD user can enter the keyword from the keyboard. The list of keywords this method will accept is set by a prior call to the ***InitializeUserInput*** method. If the user enters a string not specified in the call to ***InitializeUserInput***, AutoCAD displays an error message and tries again (and redisplays the prompt, if one was specified). If the user doesn't enter anything, but hits the ENTER key, ***GetKeyword*** returns an empty string ("") unless the call to ***InitializeUserInput*** also disallowed NULL input.

When using ***GetKeyword, GetInteger, GetReal, GetDistance, GetAngle, GetOrientation, GetPoint,*** and ***GetCorner*** keywords are active and generate an error message when detected. If using one of the above methods and there are

defined keywords using ***InitializeUserInput*** AutoCAD will generate an error with the following description:

Err.description = "user input is a keyword"

which can then be handled. At this stage the keyword is retrieved with a call to the utility method ***GetInput***[1]:

ThisDrawing.Utility.GetInput

GetInput has the form:

RetVal = object.***GetInput***()

Object	Utility
	The object or objects this method applies to.
RetVal	String
	The index specifying which keyword was entered.

This method retrieves a keyword entered by the AutoCAD user during a call to one of the user-input functions (Get*** methods). The maximum length of the keyword is 511 characters (with the 512th character reserved for the NULL character). A call to ***GetInput*** is meaningless and will fail unless it immediately follows a call to one of the user-input functions. Even then, the call will be successful only if the user-input function has returned the error description "User input is a keyword." It isn't necessary to call ***GetInput*** after a call to the ***GetKeyword*** method.

So after defining some keywords with a call to ***InitializUserInput*** it is possible to add more functionality to Get*** methods. Adding the code to what is already there from getting the escape key that is now present:

```
Private Declare Function GetAsyncKeyState Lib "user32" (ByVal vKey As Long) As Integer
Private Const VK_ESCAPE = &H1B
Dim pnt As Variant
Dim keywd As Variant
Dim userinp As Variant
Keywd = Array("case1","case2"......etc)
ThisDrawing.Utility.InitializeUserInput 1,keywd
On Error Resume Next
pnt = ThisDrawing.Utility.GetPoint (prevpnt , "next point : ")
If Err Then
  If GetAsyncKeyState(VK_ESCAPE) And &H8000& Then ' (The Esc key was selected)
```

```
End If
  If StrComp(Err.Description, "user input is a keyword", 1) = 0 Then
   userinp = ThisDrawing.Utility.GetInput
   If userinp = "case1" Then

  Else
   If usernp = "case2" Then

   End If
  End If
 End If
 Err.Clear
End If
```

Note: The ***StrComp***[1] method is provided by the VBA Strings object and is used for comparing strings. It has the form:

Rtnval = ***StrComp***(string1, string2[, compare])

string1 Required. Any valid string expression.

string2 Required. Any valid string expression.

compare Optional. Specifies the type of string comparison. The compare argument can be omitted, or it can be 0, 1 or 2. Specify 0 (default) to perform a binary comparison. Specify 1 to perform a textual comparison. For Microsoft Access only, specify 2 to perform a comparison based on information contained in the database. If compare is Null, an error occurs. If compare is omitted, the Option Compare setting determines the type of comparison.

Rtnval

If	StrComp returns
string1 is less than string2	-1
string1 is equal to string2	0
string1 is greater than string2	1
string1 or string2 is Null	Null

4.5 Transparent Commands

Finally when requesting input from the user, deal with what happens when the user selects a transparent command. A transparent command is one executed while another command is in progress. They would include commands such as zoom, zoom window etc. It is important that these commands work during, for example a call to ***GetPoint***, as the user will invariably want to move around the graphics screen. The **On Error Resume Next** statement will basically look after these commands i.e. the use of the transparent command will generate an error and if the flow is allowed to continue with the **On Error Resume Next** statement the transparent command will execute. Although the command executes the call to the ***GetPoint*** (or any other Get*** method) has failed and it is needed to loop back and call it again for valid input. This can be done with the use of a boolean variable and a **Do While** loop.

This can be added to the code above as follows:

```
Dim transparency As Boolean
Keywd = Array.........
 On Error Resume Next
  Do
    transparency = False
    pnt =  ThisDrawing.Utility.GetPoint (prevpnt , "next point : ").
'etc
                         Else              ('If StrComp(Err.Description..............)
                           transparency  = True
                         End If
                    Err.Clear
                End If
  Loop While transparency
```

A pretty robust loop is now present for getting input from the user. In defining the floor area the user needs to loop through this and keep getting the vertices of the floor until such time as all the vertices have been covered. Two keywords were defined which the user will need when defining the floor area. These keywords are as follows "c" for close and "u" for undo last entry. Each time the user selects a

point from the graphics screen it is added to an already created array of coordinate points. If the user types "u" for undo the last coordinate added to this array is erased and the user re-prompted for the next point. The loop keeps executing until such time as the user chooses "c" for close. This basically means close the floor area and add a polyline to the drawing with the coordinates selected by the user. It was endeavored to keep the functionality of the Esc key to behave as close as possible to its behavior within the AutoCAD environment i.e. just quit the VBA routine when the Esc key is pressed. As well as adding the newly created floor area (as an AutoCAD polyline) to the drawing's Model Space it is also added to a block called "floor area" so that it can be easily retrieved at a later stage.

Chapter 5 Programming Techniques

5.1 Adding A Block To The Blocks Collection

There are three different types of block: the simple block, the XRef block, and the layout block. A simple block is a collection of objects that can be associated together to form a single object, or block definition. It is possible to insert, scale, and rotate a simple block in a drawing. It is possible to explode a simple block into its component objects, modify them, and redefine the block definition. Simple blocks can be defined from geometry in the current drawing, or by using another AutoCAD drawing. When using another drawing to create a block, the new simple block is stored in the current drawing database, and therefore it is not updated if the original drawing changes. Inserting an instance of a simple block into the current drawing creates a **BlockReference** object.

An XRef block is an external link from another drawing to the current drawing. Because the XRef block represents a link to geometry, not the geometry itself, it is updated whenever the original drawing changes. Inserting an instance of an XRef block into the current drawing creates an **ExternalReference** object.

In order to create a block programmatically use the following method:

Dim flr As AcadBlock

Set flr = ThisDrawing.Blocks.Add [1](InsertionPoint, "FloorOutline")

where the method's arguments are as follows:

InsertionPoint 3-element array of doubles defining base point from which block will be inserted

Name in this case the block is named - "FloorOutline"

When naming the block be aware of illegal characters in the block name. Special characters that cannot be used include less-than and greater-than symbols (< >), forward slashes and backslashes (/ \), quotation marks ("), colons (:), semicolons (;), question marks (?), commas (,), asterisks (*), vertical bars (|), equal signs (=), and back quotes ('). Sometimes these characters will not be picked up when

naming blocks programmatically and will only be picked up the next time the drawing is audited during which AutoCAD will change their names to acceptable ones. Since it was never intended to insert the "FloorOutline" block into the drawing its InsertionPoint is not significant so define an array and assign arbitrary double values to the individual elements of the array:

```
Dim InsertionPoint (0 To 2) As Double
InsertionPoint (0) = 0 : InsertionPoint (1) = 0 : InsertionPoint (2) = 0
```

Now that a block has been defined in the drawing that is ready to have objects added to its definition the polyline can be added defining the floor outline to it. It can also be added to the **Model Space** (which is basically a block anyway).

5.2 Adding Objects To A Block

Define the floor area using a **LightweightPolyline** Object that is a 2-D line with adjustable width composed of line and arc segments (Acad object name: AcadLWPolyline). In this case it will just have line segments (between vertices) whose width are zero. In order to create the **LightweightPolyline** Object the following method can be used: object.***AddLightweightPolyline***(VertexList) where object can be **Model Space, Paper Space or Block**. In this case add this to both the newly created block and the drawing's Model Space as follows:

```
Dim floorarea As AcadLWPolyline
Set floorarea = flr.AddLightweightPolyline (VertexList) 'object added to block
Set floorarea = ThisDrawing.ModelSpace.AddLightweightPolyline (VertexList)
```

The VertexList parameter is an array of doubles specifying the x and y coordinate values of each individual vertex. Since the array is 0 indexed an array element with an odd number implies the x coordinate while an even number implies the y coordinate. Remembering that the vertices information was collected using the ***GetPoint*** method it is now necessary to process that information collected to suit ***AddLightweightPolyline*** method. From ***GetPoint*** there is an array each individual element of which is a 3 element array of doubles specifying an x, y and z coordinate. The x and y coordinates are the only ones of interest. As each vertex is collected a count variable can be used to give the overall number of vertices in the

floor area. Armed with this variable and the coordinate array the information can now be processed to produce a VertexList:

pnt = array of vertex coordinates (retrieved from user)

count = number of vertices (evaluated during execution of loop)

```
Dim VertexList (0 To count*2 + 1)
Dim i As Integer, j As Integer, k As Integer
i = 0 : j = 1 :k = 0
  While k <= count – 1
   VertexList(i) = CLng(pnt(k)(0))   'x coordinate
   VertexList(j) = CLng(pnt(k)(1))   'y coordinate
   i = i + 2 : j = j + 2 : k = k + 1    'move to the next array element
 Wend
VertexList(i) = CLng(pnt(0)(0))      'polyline is closed therefore first = last coord
VertexList(j) = CLng(pnt(0)(1))
```

CLng coerces the value in brackets to a long integer. Since AutoCAD deals with real values. It is a good idea to round off the user input at this stage. It will make de-bugging hopefully a little bit easier later on when comparing numbers.

At this stage of the development, how to get the coordinates of the newly created floor area over to the Delphi environment for processing, was considered. The first approach was as stated earlier to use database tables.

5.3 Connecting A Project To A Database

Database tables were the preferred method of connecting AutoCAD to the Delphi environment in the early stages of this research and development. During the research they were found to be inefficient as a mere transient data link between AutoCAD and Delphi. They were being used as an intermediary storage container for data transferring back and forth from AutoCAD to Delphi. After some research a more efficient method became apparent and since the research followed this new approach the database table method was abandoned. When connecting to the database in the following discussion it is assumed that Paradox tables underlie the database configuration. First these database tables were created to contain the required information by the project. In creating these tables (using Database

Desktop provided with the Delphi 5 environment) fields of a specific data type were defined into which coordinate information was written to and read from. These tables are then saved to a folder from where they can be accessed by the database. Next the database needs to be configured using the Windows 32 bit Open Database Connectivity (ODBC) Interface provided by Microsoft.

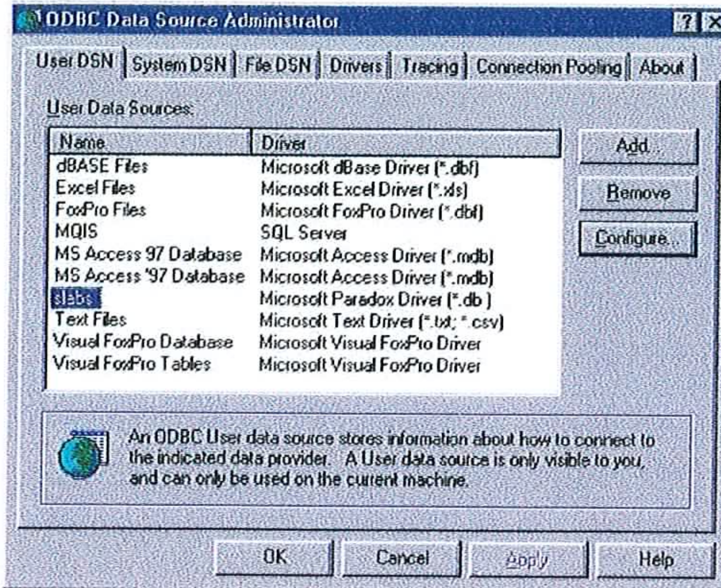


Fig 5.3.1 The ODBC Administrator

Microsoft's ODBC Administrator is accessed from Control Panel and provides an interface that allows a database to be configured for use by AutoCAD.

To configure a database accessible from AutoCAD the Add button in the ODBC Data Source Administrator is selected. Depending on which tab is selected the data source will have varying levels of accessibility – a data source which most suits the application is chosen. When choosing the Add button a prompt will be given to select a data source type to add to the already configured data sources. On selecting the appropriate data source type a request is then made to give the new data source a name and in the case of Paradox a version number.

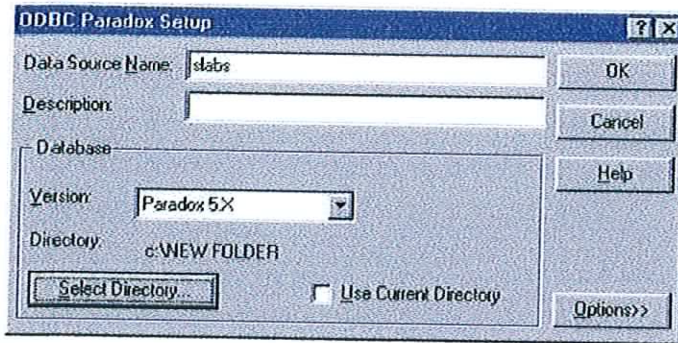


Fig 5.3.2 Setting up a Database

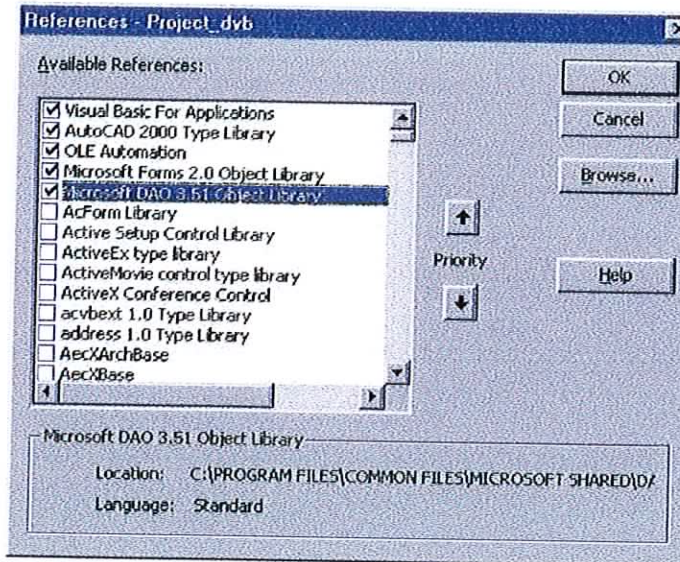
A name and description for the data source is chosen. The version number is then selected, the "use current directory box" un-checked and the "select directory" button clicked. On selecting the directory the folder in which the database tables created earlier were saved is pointed to.

AutoCAD is now ready to connect to the database.

objects of the database to the AutoCAD VBA environment a reference is set to the Microsoft Data Access Objects (DAO) Library. . Once a reference has been set to the DAO object library the DAO objects can be viewed in the Object Browser by clicking DAO in the Project/Library box.

From the Tools menu if references is chosen:

AutoCAD displays the references dialogue box.



If the Microsoft DAO Object Library is not one of the available references it can be installed from the Microsoft Office disk. It is one of the options which can be checked during a customised installation of the Office product.

Fig 5.3.3 Available Project References

5.4 Implementing The Database With DAO

Microsoft DAO provides a way to control a database from any application that supports Visual Basic for Applications. Some DAO objects represent the structure of the database, while others represent the data itself. By using DAO, local or remote databases can be created in a variety of formats, and work with their data. With DAO objects;

- Create a database or change the design of its tables, queries, indexes, and relationships.
- Retrieve, add, delete, or change the data in the database.
- Implement security to protect the data.
- Work with data in different file formats and link tables in other databases to the database.
- Connect to databases on remote servers and build client/server applications.

DAO can be used to work with databases in different formats. There are three different categories of database formats that are accessible through DAO. The first type of format is the Microsoft Jet format. DAO can be used to work with all databases created with the Microsoft Jet database engine.

The second type of database format is the installable ISAM format. An installable ISAM is a driver that provides access to external database formats through DAO and the Microsoft Jet database engine.

The third type of database format that is accessible through DAO is an ODBC data source that is what was previously configured. If working with an ODBC data source, DAO operations can be processed through Microsoft Jet, or ODBCDirect can be used to circumvent the Microsoft Jet Engine and work directly with the data in the ODBC data source. Whether using DAO with Microsoft Jet or with ODBCDirect to work with an ODBC data source depends on what kinds of operations are to be performed on the data source.

Because not all DAO features are available with ODBCDirect, Microsoft DAO still supports ODBC through the Microsoft Jet database engine. ODBC can be used through Microsoft Jet, ODBCDirect, or both, with a single ODBC data source. Which of these two methods to use to access an ODBC data source is determined

by what type of workspace is being worked in. A workspace, represented by a Workspace object, is an active session for a particular user account. A session marks a sequence of operations performed by the database engine. A session begins when a particular user logs on and ends when that user logs off. The operations that a user can perform during a session are determined by the permissions granted to that user.

DAO objects in code are referred to in the same way that other objects are referred to. Objects that represent the structure of the database are saved with the database. Objects that are used to work with the data in the database generally are not saved, but are created each time they are needed

When a new DAO object is created to be saved with the database, it must be appended it to the appropriate collection of saved objects by using that collection's Append method. To connect the project to this ODBC data source the following code can be used:

```
Dim wkspace As Workspace
Dim dbase As Database
Set wkspace = CreateWorkspace ("", "", "", dbUseODBC)
Set dbase = wkspace.OpenDatabase ("slabs", dbDriverNoPrompt, False)
```

Newly created Workspace objects — those created with the ***CreateWorkspace***[4] method — are not automatically appended to the **Workspaces** collection. The Append method of the **Workspaces** collection can be used to append a new **Workspace** object if it is to be part of the collection. However, the **Workspace** object can be used even if it's not part of the collection.

Append the new **Workspace** object to the **Workspaces** collection if it is required to use the **Workspace** from procedures other than the one in which it was created in.

CreateWorkspace has the following form:

```
Set wkspace = CreateWorkspace(name, user, password, type)
```

wkspace	An object variable that represents the Workspace object to be created.
Name	A String that uniquely names the new Workspace object
User	A String that identifies the owner of the new Workspace object.

password	A String containing the password for the new Workspace object. The password can be up to 14 characters long and can include any characters except ASCII character 0 (null).
type	Optional. A constant that indicates the type of workspace, as described in Settings.

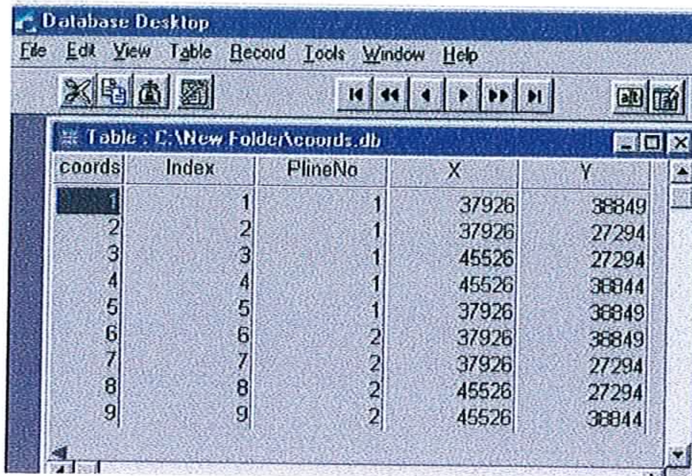
The Database object represents an open database. It can be a Microsoft Jet database or an external data source. The Databases collection contains all currently open databases.

OpenDatabase[4] has the following form:

Set dbase = wkspace.OpenDatabase (dbname, options, read-only, connect)

dbase	An object variable that represents the **Database** object to be opened.
wkspace	Optional. An object variable that represents the existing **Workspace** object that will contain the database. If a value for workspace is not included, ***OpenDatabase*** uses the default workspace.
dbname	A String that is the name of an existing Microsoft Jet database file, or the data source name (DSN) of an ODBC data source. (If opening a database through an ODBCDirect workspace and the DSN in connect is provided, dbname can be set to a string of choice that can be used to reference this database in subsequent code.)
options	Optional. A Variant that sets various options for the database. (dbDriverNoPrompt: The ODBC Driver Manager uses the connection string provided in dbname and connect. If sufficient information is not provided a run-time error occurs.).
read-only	Optional. A Variant (Boolean subtype) value that is True if the database has read-only access, or False (default) if the database has read/write access.
connect	Optional. A Variant (String subtype) that specifies various connection information, including passwords.

When the database is open the table is ready to populate the x, y coordinates of the floor area. The database contains a table with the following structure:



coords	Index	PlineNo	X	Y
1	1	1	37926	38849
2	2	1	37926	27294
3	3	1	45526	27294
4	4	1	45526	38844
5	5	1	37926	38849
6	6	2	37926	38849
7	7	2	37926	27294
8	8	2	45526	27294
9	9	2	45526	38844

Fig 5.4.1 A Database Table

The table can be populated by means of **Recordsets**. The **Recordset** object represents a set of records within the database. The **Recordsets** collection contains all open **Recordset** objects. DAO offers five types of **Recordset** objects: table-type, dynaset-type, snapshot-type, forward-only-type, and dynamic-type. **Dynamic-type Recordset**[4] objects were used which are available only in ODBCDirect workspaces.

Dynamic-type Recordset objects are updated dynamically as other users make modifications to the underlying tables. **Dynamic-type Recordset** objects are available only if using an ODBC driver that supplies its own cursors. Because not all ODBC drivers supply their own cursors, it is necessary to determine whether it does before trying to open a **Dynamic-type Recordset** object. If the ODBC driver doesn't supply its own cursors, then a snapshot-type or forward-only-type **Recordset** object can be used instead. The advantage of using a **Dynamic-type Recordset** object is that the Recordset will immediately reflect any changes to the data, including added or deleted records.

For example, if a **Dynamic-type Recordset** object is opened and another user edits a record in one of the underlying tables, that change will be reflected in the **Recordset** opened. In order to do this, however, DAO must constantly re-query the data source, which may slow performance considerably. Therefore, avoid using **Dynamic-type Recordset** objects except in situations where it's crucial to have the most up-to-date data at all times.

Using the table structure referred to above it is possible to now open the following **Recordset**:

Dim coords As Recordset

Set coods = dbase.OpenRecordset[2]("coords", dbOpenDynamic, dbRunAsync, dbOptimistic)

where ***OpenRecordset*** takes the following arguments;

source A String specifying the source of the records for the new **Recordset.** The source can be a table name, a query name, or an SQL statement that returns records. For table-type **Recordset** objects in Microsoft Jet databases, the source can only be a table name. ("coords" = table name)

type Optional. A constant that indicates the type of **Recordset** to open. (dbOpenDynamic opens a dynamic-type Recordset object, which is similar to an ODBC dynamic cursor.)

options Optional. A combination of constants that specify characteristics of the new **Recordset.** (dbRunAsync runs an asynchronous query which is a type of query in which SQL queries return immediately, even though the results are still pending. This enables an application to continue with other processing while the query is pending completion.)

lockedits Optional. A constant that determines the locking for the **Recordset.** (dbOtimistic refers to a type of locking in which the data page containing one or more records, including the record being edited, is unavailable to other users only while the record is being updated by the Update method, but is available between the Edit and Update methods. Optimistic locking is used when accessing ODBC

databases or when the LockEdits property of the Recordset object is set to False.)

Having now opened the **Recordset** the x, y coordinates can be appended to the coords table:

```
Dim vertices As Variant
Dim flr As AcadBlock
Dim flooroutline As AcadLWPolyline
Set flr = ThisDrawing.Blocks.Item("FloorOutline") '(Block name given earlier)
Set flooroutline = flr.Item(0) (its the only item in the block)
vertices = flooroutline.Coordinates 'returns a variant array of doubles defining the x,y coordinates of the vertices of the polyline
K = 0
 While k <= (UBound(vertices) - 1)
   coords.AddNew    'Add new records to the recordset object
   coords.Fields("x").Value = vertices(k)
   k = k + 1
   coords.Fields("y").Value = vertices(k)
   coords.Update    'send the new data out to the database table.
   k = k + 1
 Wend
coords.Close
dbase.Close          ' closes the database
```

The coordinate data has now been exported to a database from where it can be accessed by the Delphi programme. On examination of the preceding code it can also be seen how AutoCAD objects could be created by reading in data from an external database. A two-way link between AutoCAD and Delphi has now been created. In the initial stages of development with the knowledge that the link existed simultaneous development continued separately in both Delphi and AutoCAD. When setting the reference to the DAO object model all the available references AutoCAD could use became apparent. This prompted an investigation into what could be done with Excel from within the AutoCAD environment.

5.5 Automating Applications With AutoCAD

AutoCAD can be used to automate any application which supports COM. Applications that can be automated will appear in the references dialogue box of the tools menu. To have control of an application first create an instance of their appropriate application objects. There are two approaches to use to achieve this

- Late Binding
- Early Binding

When using late binding a reference is not set to the application's object library in the references dialogue box. The variable used to create the application object will not take on any of the methods or properties of the application until runtime and there won't be any type checking of parameters etc. at compile time. The code completion feature of the VBA IDE also will not work on the application objects and the objects and their associated help files will not be available in the Object Browser. To get a late bound reference to Microsoft Excel the following code can be used:

```
Dim Excel As Object
On Error Resume Next
 Set excel = GetObject(, "Excel.Application")
 If Err <> 0 Then
   Err.Clear
   Set excel = CreateObject("Excel.Application")
     If Err <> 0 Then
       MsgBox "Could not load Excel.", vbExclamation
     End If
 End If
On Error GoTo 0
```

GetObject[1] expects an instance of the application to be running on the desktop and takes two arguments:

pathname	Optional; Variant (String). The full path and name of the file containing the object to retrieve. If pathname is omitted, class is required.
class	Optional; Variant (String). A string representing the class of the object. The class is the formal definition of an object. The class

acts as the template from which an instance of an object is created at run time. The class defines the properties of the object and the methods used to control the object's behavior. The class argument uses the syntax appname.ObjectType and has these parts:

appname	Required; Variant (String). The name of the application providing the object.
ObjectType	Required; Variant (String). The type or class of object to create.

If there is no instance of the application running on the desktop create an instance using ***CreateObject*** and the class of the application to create.

Note: the **On Error Goto 0** statement is used to disabled any enabled error handlers in the current procedure. Early binding was the preferred method. As well as giving better performance the VBA IDE code completion feature will work for objects and their associated help files can be viewed in the Object Browser Window. To access the application object using early binding first check a reference to the application object in the references dialogue box of the tools menu. For Excel set the reference to Microsoft Excel 8.0 object library. Then dimension variables to be application specific:

```
Dim Excel As Excel.Application
Dim Excelsheet As Worksheet
On Error Resume Next
Set excel = GetObject(, "Excel.Application")
'etc with CreateObject
On Error GoTo 0
```

With a reference to the application object and a knowledge of its object 'model whatever automation is necessary can be done:

```
Excel.Visible = True
Excel.Workbooks.Add
Excel.Sheets("Sheet1").Select
Set Excelsheet = Excel.ActiveWorkbook.Sheets("Sheet1")
Excelsheet.cells(count, 1).Value = vertices(k)
'Etc.
```

At this stage the real power of COM was becoming apparent. It was realised that just as other applications can be automated from within AutoCAD it followed that AutoCAD could be automated from within another application. Delphi programmes could be shown to open AutoCAD and draw some entities. With this accomplished the task of using the Delphi programme to open AutoCAD and prompt the user for the required information regarding floor outlines etc was set about. This was to render the use of database tables obsolete and was to be a direct link between AutoCAD and Delphi. This works quite well (the Delphi development is discussed in the 2nd section of this document) except that AutoCAD has now lost the focus and Delphi has taken control of the environment. Experienced technicians in the course of producing floor layout drawings would be reluctant to leave the AutoCAD environment and would favour a tool that works from within AutoCAD. What was needed was a Delphi programme that would lie dormant until such time as the user needed it. When required it would jump into action perform whatever task was needed of it and then lie in wait until such time as it was called upon again. Anybody with any programming experience would recognise this situation as one in which to employ the use of a dynamic link library (DLL).

Chapter 6 The New Approach

6.1 Dynamic Link Libraries – A Brief Introduction

The intricacies of dynamic link libraries and their construction will be discussed in detail in the second section of this document. DLLs are modules that contain functions and data. A DLL is loaded at run time by its calling module (in this case acad.exe). When a DLL is loaded, it is mapped into the address space of the calling process.

DLLs can define two kinds of functions: exported and internal. Other modules can call the exported functions. Internal functions can only be called from within the DLL where they are defined. Although DLLs can export data, its data is usually only used by its functions. DLLs provide a way to modularize applications so that functionality can be updated and reused more easily. They also help reduce memory overhead when several applications use the same functionality at the same time, because although each application gets its own copy of the data, they can share the code.

With the use of a DLL there is no need to leave the AutoCAD environment. The idea now is to create a toolbar in AutoCAD that will pre-process all the floor slab information. Once this is done the information can be sent to the DLL that will further process it into individual slab elements that can be manufactured. This concept is most easily understood if the newly created toolbar and its individual tools are examined.

6.2 Programmatically Creating A Toolbar

As toolbar objects did not appear in the AutoCAD R 14.01 object model the operation of a toolbar and its buttons was mimicked with the use of a UserForm and command buttons. This approach was abandoned with the introduction of the toolbar object to the AutoCAD 2000 object model. Toolbars can now be created with VBA code. The definitions including the appearance, associated macros, associated bitmaps etc. are stored in AutoCAD menu files. The file that stores all the toolbar information is called acad.mns and is a text file (it can be found under

the ***Toolbars header for all the toolbar information) located in the AutoCAD directory (usually the support sub-directory). AutoCAD does not use this file directly but compiles it into two files with the same name but the extensions .mnr and .mnc respectively. The file with the extension .mnr is a resource file that contains all the bitmap information required by the various toolbars and the file with the extension .mnc is the compiled menu file.

A new toolbar can be added to an existing menu file or else have a new menu file created for it. New (empty) menu groups cannot be created programmatically. However, an existing menu file can be loaded containing a menu group and saved out again with a new menu group name and to a new menu file. The new menu group can then be edited to contain the menus and toolbars desired.

The first step in creating a new toolbar is to get reference to the **MenuGroup** Object. A menu group contains menus and toolbars, some or all of which may be currently displayed in AutoCAD. Each menu group contains a **PopupMenus** collection and a **Toolbars** collection. The **PopupMenus** collection contains all the menus within the menu group and can be accessed through the Menus property. Likewise, the **Toolbars** collection contains all the toolbars within the menu group and can be accessed through the Toolbars property.

To add a new toolbar to the Acad menu group:

```
Dim mybar As AcadToolbar
Dim mnugrp As AcadMenuGroup
Dim acad As AcadApplication
Set acad = ThisDrawing.Application
Set mnugrp = acad.MenuGroups.Item("acad") ' gets the acad MenuGroup
Set mybar = mnugrp.Toolbars.Add("New Toolbar") ' the new toolbar name in quotes
```

At this stage applying the SaveAs method to the MenuGroup Object can create the new menu group:

```
mnugrp.SaveAs "New Menu Name",filetype
```

where the new menu name is quoted and it can be saved as file type:

acMenuFileCompiled:	A compiled menu file (MNC file type).
acMenuFileSource:	A source menu file (MNS file type).

A new toolbar object has now been created to which buttons, macros and bitmaps can be added.

Before adding any toolbuttons the toolbar needs to be visible:

mybar.Visible = True

To add toolbuttons:

Dim mybutton As AcadToolbarItem

Set mybutton = mybar.AddToolbarButton(0, "Get_Slab_Boundary", "", Chr(3) & Chr(3) & "_-vbarun getoutlines ")

where ***AddToolbarButton*** [1] has the following format:

mybutton = mybar.***AddToolbarButton***(Index, Name, HelpString, Macro[, FlyoutButton])

mybar AcadToolbar

The object or objects this method applies to.

Index Variant; input-only

The index must be either an integer or a string. If an integer, the index must be between 0 and N-1, where N is the number of objects in the toolbar. The new item will be added immediately before the specified index location.

Name String; input-only

The string that identifies the toolbar button. The string must be comprised of alphanumeric characters with no punctuation other than a dash (-) or an underscore (_). This string is displayed as the tooltip when the cursor is placed over the toolbar button.

HelpString String; input-only

The string that appears in the AutoCAD status line for the button.

Macro String; input-only

The command associated with this item.

FlyoutButton Variant; input-only; optional

A boolean variable stating if the new button is to be a flyout button or not. If the button is to be a flyout button, this parameter must be set to TRUE. If the button is not to be a flyout button, this parameter can be set to FALSE or ignored and the new ToolbarItem object will be

returned. (A flyout button is one that has a black triangle in the bottom right hand corner. When the left mouse button is held down over the button an extra drop down list of buttons appears each of which does an individual task)

When the toolbutton has been created it is then possible to add icons onto the button to give it character.

mybutton.SetBitmaps "Small Icon Name", "Large Icon Name"

Where small and large icon names can be either the names of any of the icons used by AutoCAD's Acad menu file (resource bitmaps stored in acadbtn.dll) or the path to any user defined bitmap. A user-defined bitmap must be of the proper size for the small parameter (16 pixels wide by 15 pixels high) and must reside in the library search path. For the big parameter, if the specified bitmap is not 24 pixels × 22 pixels, AutoCAD scales it to that size. A user-defined bitmap with the file name and .bmp extension needs to be specified.

The macro property of the button can be used to run a VBA macro with the following syntax:

Chr(3) & Chr(3) & "_-vbarun getoutlines "

This assigns the command ^C^C_-vbarun getoutlines to the toolbutton which is the equivalent of pressing the Esc button twice and running the VBA macro called getoutlines. The syntax here is important leaving a space between the end of the macro name and the closing quotes. A VBA macro is a sub-routine that once it is loaded it is exposed to the user via the macros dialogue box

Tools-Macro-Macros

Command : vbarun

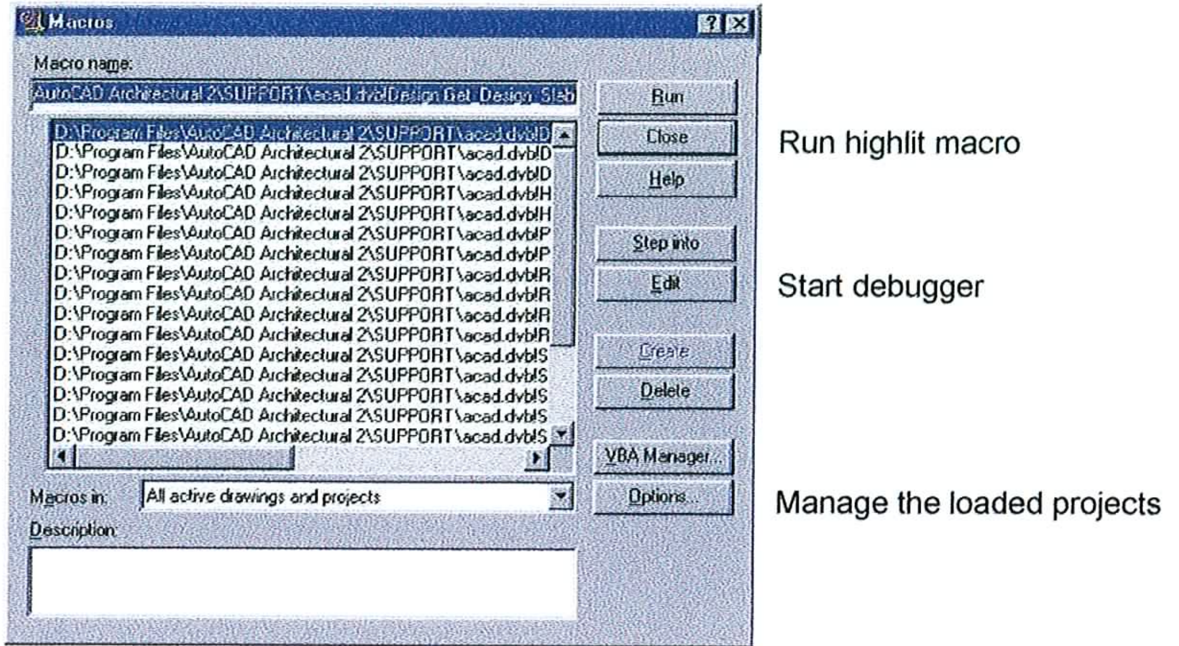


Fig 6.1.1 Available Macros Dialogue Box

Once the new toolbar has been created it is possible to dock it using the dock procedure:

mybar.Dock acToolbarDockTop

where acToolbarDockTop can be replaced with any of the following: acToolbarDockBottom, acToolbarDockLeft or acToolbarDockRight.

6.3 Adding A Macro To The Project

Macros reside as Public Subs in modules or in the Drawing Object code of the project. Modules are used to break the code up into more easily managed units. Modules come in two forms:

- Standard Module
- Class Module

A standard module contains only procedure, type, and data declarations and definitions. Module-level declarations and definitions in a standard module are Public by default. A class module contains the definition of a class, including its methods and properties. To insert a macro into the project choose Insert-

Procedure from the tools menu. AutoCAD displays the Add Procedure dialogue box.

A Public Sub added to the module or the drawing code will be available as a macro from the command line or the macros dialogue box. A function will return a value to the code while a property describes a characteristic of an object within the code.

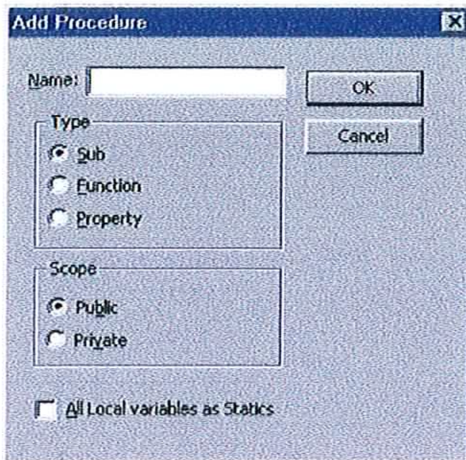


Fig 6.2.1 Adding a Procedure

Once the macro has been inserted into the VBA code it can be accessed in various ways from the AutoCAD environment. Using a toolbar to access the macros is the simplest interface to the VBA code.

Chapter 7 The Toolbar

7.1 A Description Of The New AutoCAD Toolbar

Using the code described above a new toolbar was created for AutoCAD that processes drawings into slab layout drawings. The toolbar looked like this:

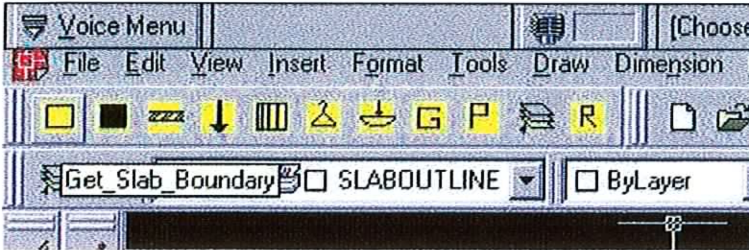


Fig 7.1.1 New Toolbar showing Tooltip for first button.

There were eleven toolbuttons on the new toolbar. From left to right they were as follows:

1. Get_Slab_Boundary
2. Get_Any_Holes
3. Get_Any_Walls
4. Point Loads
5. Get_Slabbed_Area
6. Hangers
7. Design
8. Group Slabs
9. Production Sheets
10. Toggle Layers
11. Reset Drawing

7.2 Get_Slab_Boundary

When processing a drawing like the one in the sample project this is the first button that should be chosen. It accesses a VBA macro called getoutlines that prompts the user to describe the outer extremities of the floor area to be slabbed by choosing the vertex points with the mouse. When using this toolbutton there are four different options available:

1. Choose a valid point with the mouse
2. Type "u" at the keyboard to undo last entry
3. Type "c" at the keyboard to close the floor area and finish selecting vertices
4. Use the Esc key to quit the macro and return the drawing to its original.

As soon as the button is selected the drawing is organised to receive the desired input. The first thing to do is to create some block entities that will store the incoming information. Any individual floor area will contain information on the following entities:

- floor outline extremities
- holes within floor area
- walls imposing on floor area
- point loads imposing on floor area

To store this information the following blocks are created :

```
Set blck = ThisDrawing.Blocks.Add(inspoint1, "polyline_block_container")
Set blck = ThisDrawing.Blocks.Add(inspoint1, "hole_block_container")
Set blck = ThisDrawing.Blocks.Add(inspoint1, "wall_block_container")
Set blck = ThisDrawing.Blocks.Add(inspoint1, "point_block_container")
```

Having done this the object snap is set variable so that the desired snap options are turned on in the drawing:

```
ThisDrawing.SetVariable "OSMODE", 2179
```

All drawing revisions are to be done on a new layer so that the newly created entities can be examined without interference from other layers. To do this a new layer needs to be created, adding it to the drawing and making it the active layer:

```
Dim layr As AcadLayer
Set layr = ThisDrawing.Layers.Add("SLABOUTLINE")
ThisDrawing.ActiveLayer = layr
```

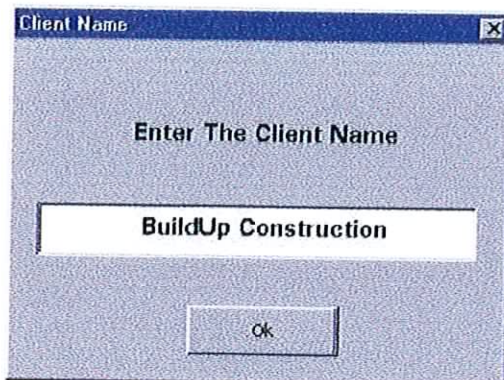
Now the vertices of the floor outline can be input as described earlier. When finished describing the floor outline the processed coordinates can be added as an **AcadLightweightPolyline** and as an **AcadRegion** (discussed in the section on the second toolbutton).

Each individual floor area will have its own physical properties:

- It belongs to a client
- It is of a particular unit size
- It has a particular screed depth
- It has an area loading

Since all the floor areas within a particular drawing will belong to the same client the user needs only to be asked once for the client detail information. With the statement UserForm1.Show

The user can be prompted to give this information with a UserForm:



On clicking OK a block called "Client Details" is created and the string value entered by the user is stored as an **AcadAttribute** within this block.

Fig 7.2.1 Client Name Dialogue Box

A UserForm is used to get the area properties unique to each floor area:

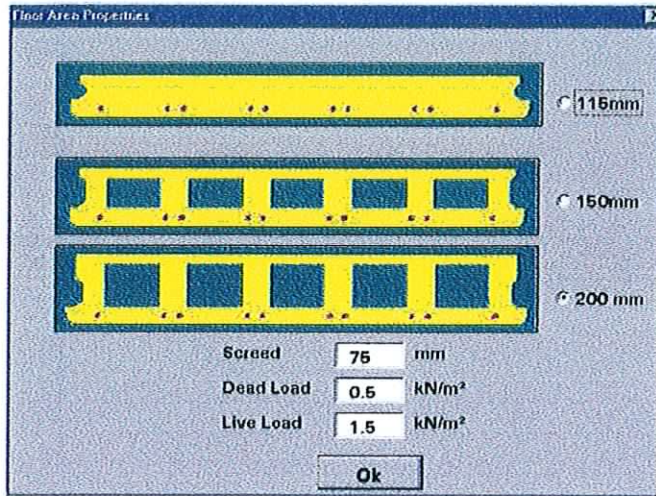


Fig 7.2.2 Floor Area Properties

On clicking OK 4 variables are set equal to the values obtained from the user inputs on the form. These variables are user variables (there are 5 integer variables UserI1-UserI5 and 5 string variables available UserS1-UserS5) of specific type which can be used anywhere in the VBA code. It is important to note that the string type variables are not saved with the drawing while the integer type variables are saved with the drawing.

Once the OK button is pressed on this form it completes the function of the first toolbutton. The code can be represented with the flowchart:

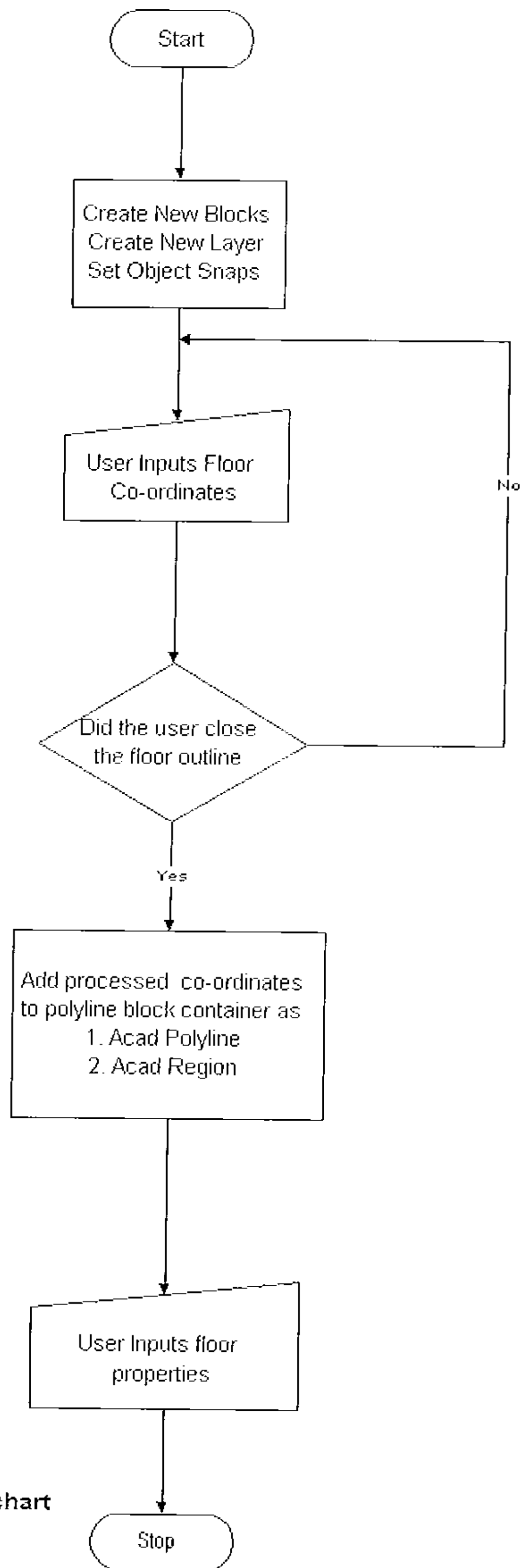


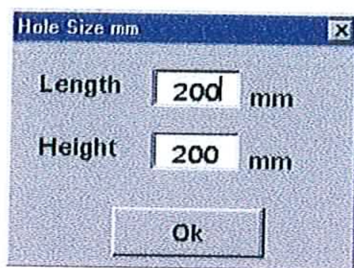
Fig 7.2.3 Define floor area flowchart

7.3 Get_Any_Holes

This button is used to locate any holes within the floor area. Holes that are wider than 400mm are not permitted and the user must be asked for a valid value. After choosing the hole size the user can only either:

1. Select a hole location
2. Choose the Esc key to quit the routine

Again the UserForm is used to get information on the hole size from the user:



Clicking OK determines the size of hole. Holes of this size are continuously added at the user defined location until the Esc key is selected

Fig 7.3.1 Hole Sizing

To determine the location of a hole with respect to the area to be slabbed two different methods can be used which can be applied to the **AcadLWPolyline** namely:

1. ***IntersectsWith***
2. ***BoundingBox***

(Note: neither of these is unique to the **AcadLWPolyline**)

The first step after getting the hole size is to convert the coordinates to an AcadLWPolyline that is initially added to the drawing's Model Space.

```
Dim hole As AcadLWPolyline
Set hole = ThisDrawing.ModelSpace.AddLightWeightPolyline (coordinates)
```

Having done this any intersection points between the floor outline and the hole outline is checked for. This is done with the ***IntersectsWith*** method. First a reference to the floor outline is needed:

```
Dim floor As AcadLWPolyline
Set Floor = ThisDrawing.Blocks.Item ("polyline_block_container").Item (0)
```

It is known that it is the first item in this block since it was added first. Secondly the region was added.

Are there any intersection points with the outline of the floor and the outline of the hole?

```
Dim intersection As Variant
intersection  = floor.IntersectWith ( hole, acextendnone)
```

where the method ***IntersectWith*** is applied to an object variable and takes two arguments as follows:

intersection = floor. IntersectWith(IntersectObject, ExtendOption)

floor	All Drawing Objects (Except Pviewport and PolygonMesh) The object or objects this method applies to.
IntersectObject	Object, input-only; The object can be one of All Drawing Objects.
ExtendOption	AcExtendOption enum; input-only This option specifies if one or the other, both, or none of the entities are to be extended in order to attempt an intersection. (acExtendNone: Does not extend either object. AcExtendThisEntity: Extends the base object. AcExtendOtherEntity: Extends the object passed as an argument. AcExtendBoth: Extends both objects.)
Intersection	Variant (array of doubles) The array of points where one object intersects another object in the drawing.

If the two objects do not intersect, no data is returned and the function UBound (Intersection) will return the value –1 else (UBound (intersection) + 1)/2 equals the number of intersection points between the two objects (remembering each intersection has an x, y and z coordinate and the array is 0 indexed).

If there are no intersection points between the two objects this must mean that the hole lies either totally outside the floor area or totally within the floor area. Using the BoundingBox property of the two objects can check this.

```
Dim maxfloorbb As Variant, minfloorbb As Variant, maxholebb As Variant
```

Dim minholebb As Variant
floor.GetBoundingBox minfloorbb, maxfloorbb
hole.GetBoundingBox minholebb, maxholebb
returns the x, y and z coordinates of a box which would encompass the complete object the box edges being parallel to the x, y and z axes of the world coordinate system.

If the hole's bounding box lies within the floor's bounding box the hole can be added as an **AcadLWPolyline** to the hole_block_container, otherwise nothing is done. If there are intersection points between the hole and the floor outline this means that only part of the hole lies within the area of the floor to the slabbed. In this case the portion of the hole that lies within the floor area is found.

When choosing a hole size for the columns in the sample project 460x460mm was chosen to allow clearance all around the column. When this hole is placed at the centre of the column the only portion of it affecting the floor area is the bottom right hand corner of the hole- essentially a hole size of 230x230.

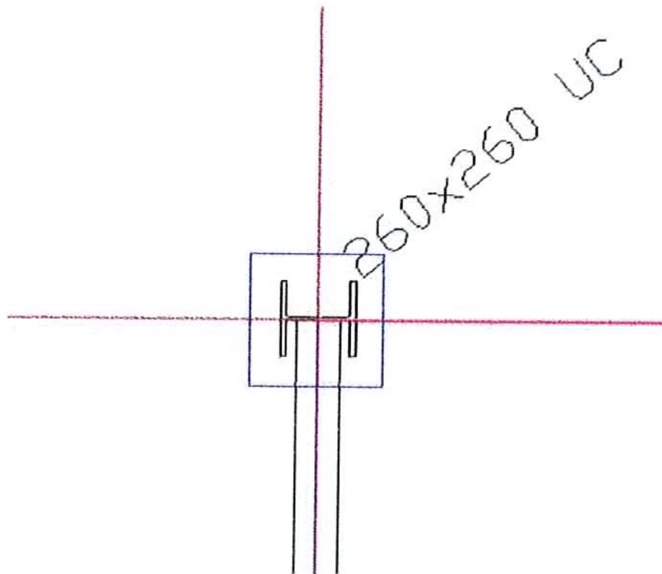


Fig 7.3.2 Resultant hole added

In this case the AutoCAD Region and particularly its Boolean and Explode methods are used to determine the new hole and the remaining floor area. Regions are two-dimensional areas created from closed shapes or loops. In this case

regions were created from closed polylines. A region is created with the ***AddRegion*** method. As shown in the following code the ***AddRegion*** method calls for an array of **AcadEntity** which must form a closed loop. These entities can be any of the following type: Line, Arc, Circle, Elliptical Arc, LightweightPolyline, and Spline.

Note ***AddRegion*** returns a variant array of Region objects and the number of objects in this array depends on the number of closed loops given in the ***AddRegion*** parameter.

RetVal = object.AddRegion(ObjectList)

Object ModelSpace Collection, PaperSpace Collection, Block

The object or objects this method applies to.

ObjectList Array of Objects

The array of objects forming the closed coplanar face to be made into a region.

RetVal Variant

This method outputs an array of the newly created Region objects.

The ***Boolean*** method is used to add and subtract regions. This will allow the determination of the portion of the hole that intersects the floor area. The ***Boolean*** method applies a mathematical algorithm to two separate regions resulting in the creation of a new region the properties of which depend upon the applied algorithm. The important thing to note here is that after application of this method one of the regions will be erased while the other will be modified:

object.Boolean(algorithm, Object)

Object 3DSolid, Region

The object or objects this method applies to.

algorithm AcBooleanType enum; input-only

acUnion: Performs a union operation. acIntersection: Performs an intersection operation. acSubtraction: Performs a subtraction operation.

Object Object; input-only

The object the operation is performed against.

The first object is modified as a result of the operation, the second is erased. Once the newly modified region is obtained the coordinates of the vertices need to be acquired. . **AcadRegion** does not support a coordinates method so it is necessary to explode the region to get at its individual elements and coordinates. The results of exploding an object in VBA depend on the object being exploded - if the structure of the base object being exploded is known it is possible to predict what the explode method will return. In this case exploding the regions will return a variant array of AutoCAD objects each element of the array corresponding to an AcadLine. By examining the StartPoint and EndPoint of each individual line in the array a new polyline can be created corresponding to the newly modified hole. The same can be done to the newly modified floor outline.

```
Dim regions As Variant
Dim holeregion As AcadRegion
Dim objectlist (0 to 0) As AcadEntity
Dim objectcollection (0 to 0) As Object
Dim floor As AcadRegion
Dim calcfloor As AcadRegion
Dim mspace As AcadModelSpace
Dim explode As Variant Set floor = ThisDrawing.Blocks.Item
("polyline_block_container").Item (1)
Set mspace = ThisDrawing.ModelSpace
Set objectcollection (0) = floor
Set calcfloor = ThisDrawing.CopyObjects ( objectcollection, mspace)
Set objectlist (0) = hole
regions = ThisDrawing.ModelSpace.AddRegion (objectlist)
Set holeregion = regions (0) 'know only one region will be created from the hole 'starting
with
```

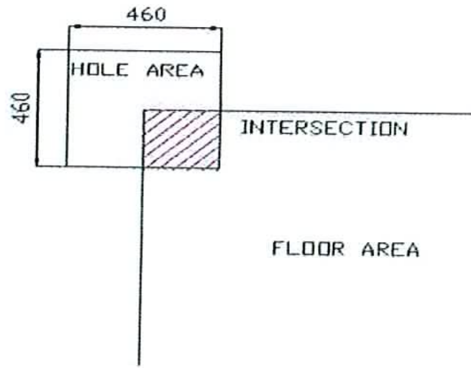


Fig 7.3.3 Intersection area between floor area and hole

holeregion.Boolean acIntersection, calcfloor

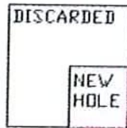


Fig 7.3.4 New hole produced

explode = holeregion.Explode *'process this array and add the new hole to the* ***hole_block_container***

floor.Boolean acSubtraction, holeregion *'produces*

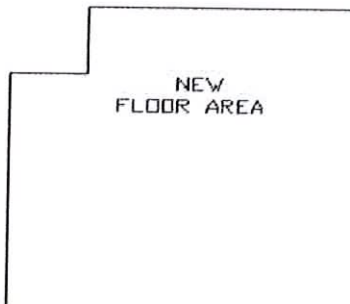


Fig 7.3.5 Floor area less new hole area

explode = floor.Explode *'this array is processed and replaces the old floor with the new in polyline_block_container.*

The CopyObjects method used above also comes in useful. During the CopyObjects operation, objects that are owned or referenced by the primary

objects in the Objects parameter will also be copied. Using this method is known as Deep Cloning[1].

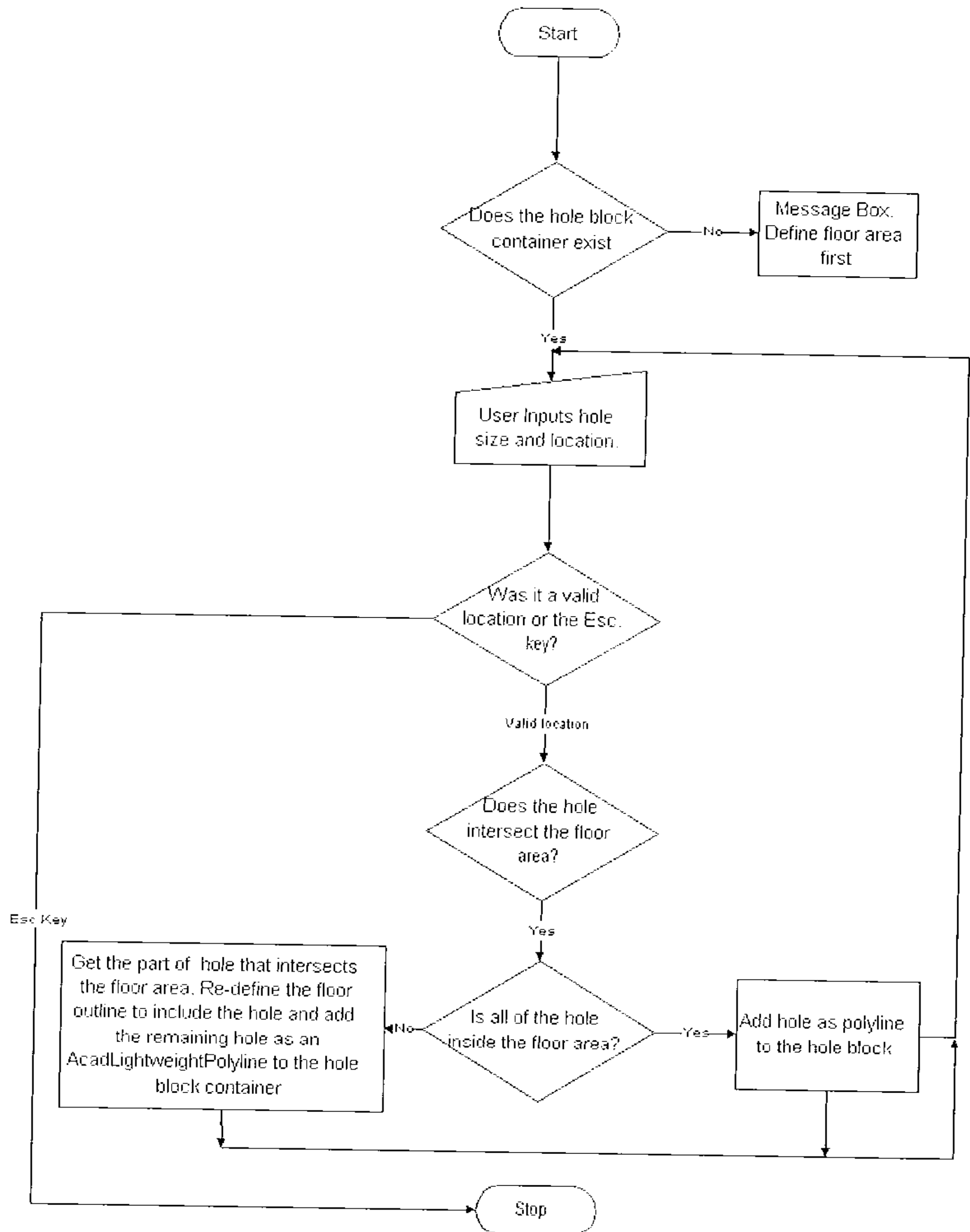


Fig 7.3.6 Hole Flowchart

7.4 Get_Any_Walls

This toolbutton uses nested blocks with attached attributes to populate a block called wall_block_container and implements the following flowchart:

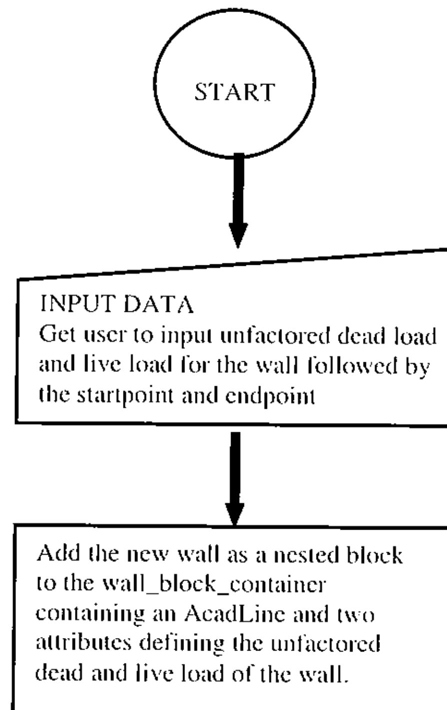


Fig 7.4.1 Wall Flowchart

7.5 Get_Point_Loads

This toolbutton uses similar concepts to the walls toolbutton and implements the following flowchart;

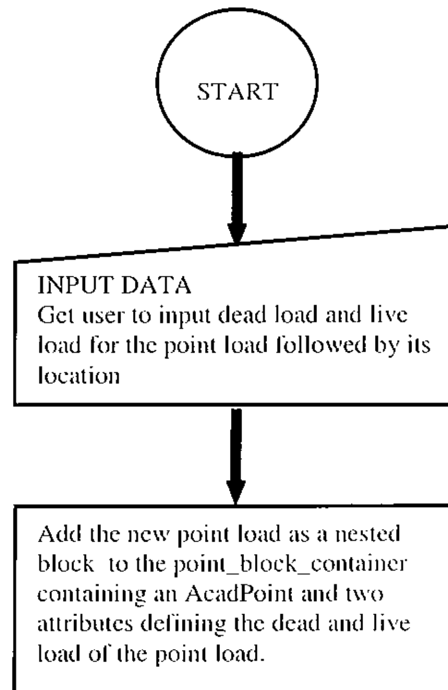


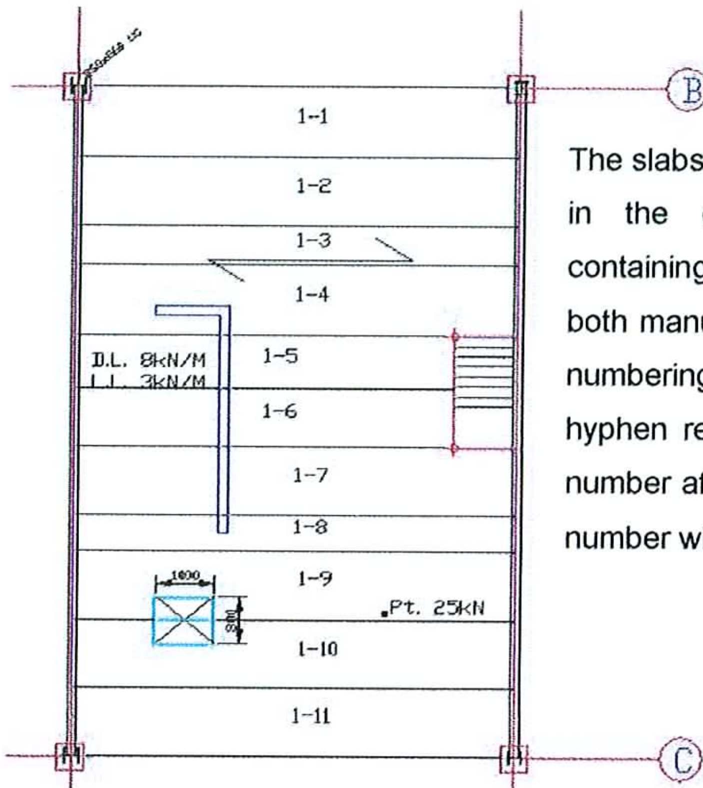
Fig 7.5.1 Point load Flowchart

7.6 Get_Slabbed_Area

The application of the first three toolbuttons completes the processing of the drawing required by the first of the DLLs the Draw.Slabs DLL. This DLL looks at the blocks containing the details for each of the floor outline, holes, walls and point loads. It then, under instruction from the user, divides the floor area up into a series of floor slabs each containing a complete picture of their own design information. The design information for each floor slab is stored in a block in the form of graphical objects and text values stored in attributes contained in the block. The block is constructed as follows:

Block Element	AutoCAD Type	Corresponds To
0	AcadRegion	Slab Boundaries
1	AcadText	Slab Identity
2	AcadLWPolyline	Hole.
No. of Holes		
Next Element	AcadLine	Wall
No. of Walls		
Next Element	AcadPoint	Point Load
No of Point Loads		
Next Element	AcadAttribute	Unit Size
+ 1	AcadAttribute	UDL Dead Load
+ 2	AcadAttribute	UDL Live Load
+ 3	AcadAttribute	Screed Depth
+ 4	AcadAttribute	Propped?
+ 5	AcadAttribute	Left Support Position
+ 6	AcadAttribute	Left Support Width
+ 7	AcadAttribute	Right Support Position
+ 8	AcadAttribute	Right Support Width
+ 9	AcadAttribute	Strand Definition
Next Elements	AcadAttributes	Wall Loads
Next Elements	AcadAttributes	Point Loads

Each block is constructed in the DLL to contain the information as structured above. Once the user has accepted a layout for the slabs this information is then written back to the AutoCAD drawing so that each individual slab has its own block structure. The method of creating the layout is discussed later but typically the sample project could be rendered into a floor slab layout looking like this:



The slabs numbered 1-1 to 1-11 are stored in the drawing as individual blocks containing all the information required to both manufacture and design them. In the numbering system the number before the hyphen refers to the floor area while the number after the hyphen refers to the slab number within that area.

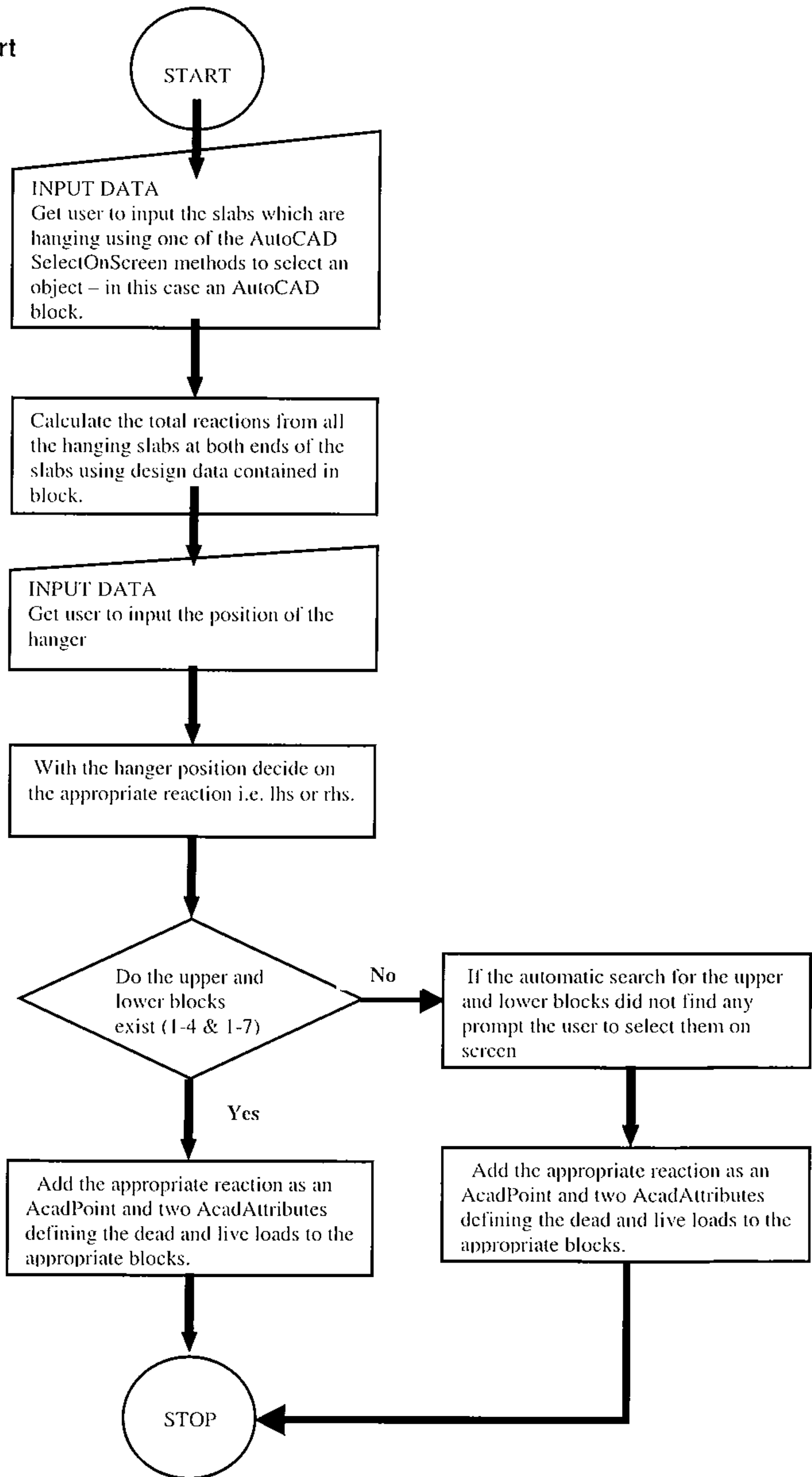
Fig 7.6.1 Slabbed floor area

On examining the above layout it is clear that the slabs numbered 1-5 and 1-6 are not supported around the stairwell. In this case a bracket known as a hanger would be made up from angle iron that would be supported from slabs 1-4 and 1-7 the length of which would support slabs 1-5 and 1-6. Consequently a point load is imposed on slabs 1-4 and 1-7 that need to be taken into account when designing these slabs. The narrow slabs are necessitated by the fact that the area to be slabbed is not a multiple of 1200mm the modular slab width. The position and width of the slabs is arrived at in the Draw.Slabs Dll.

7.7 Hangers

This toolbutton is used if what are known as hanging slabs are present described above in the drawing. This toolbutton basically adds a point load to the block structure belonging to the slab upon which the point load is imposed. When applied to the sample project this button calculates the point load imposed by slabs 1-5 and 1-6 and adds this point load to the block definitions for slabs 1-4 and 1-7. There follows a flowchart for this toolbutton:

Fig 7.7.1 Hangers Flowchart



7.8 Design

Now that a block with the complete design information ready to be extracted is complete it is possible to design any of the slabs using the design DLL. The design DLL expects a number of design variables that present themselves in the form of a variant array to the design DLL. The variant array is structured as follows:

Element	Type	Array Size	Corresponds To
0	Array	1x2	slab id & client name
1	Array	1x12	global design details
2	Array	depends on no.	wall info.
3	Array	depends on no.	point load info.
4	Array	depends on no.	hole info.
5	AcadBlock	n/a	slab to be designed

The 12 elements of global design details are:

1. Unit size
2. Screed depth
3. Whether or not the unit is propped
4. Area dead load
5. Area live load
6. Overall unit length
7. Overall unit width
8. Distance from left hand corner of the unit to the left hand support
9. Width of the left hand support
10. Distance from the left hand corner of the unit to the right hand support
11. Width of the right hand support
12. A string defining the number and position of pre-stressing strands

Wall information is passed over in the array corresponding to element No. 2. This array will either be empty in which case no walls impose on the slab or else it will contain elements each of which is an array in its own right describing an individual wall. This array contains 4 elements of information:

1. Distance from left hand corner of the slab to the start of the wall in the direction of the span
2. Distance from the left hand corner of the slab to the start of the wall perpendicular to the direction of the span
3. Distance from the left hand corner of the slab to the end point of the wall in the direction of the span
4. Distance from the left hand corner of the slab to the end point of the wall perpendicular to the direction of the span
5. Dead load imposed on the slab by the wall
6. Live load imposed by on slab by the wall

The information pertaining to the walls and the holes is passed over in a similar fashion. A reference to the block (i.e. slab) is also passed over so that the DLL can modify its information on arrival at an acceptable design.

The design module will be discussed more in-depth later but it basically has the ability to change any of the following in order to produce an acceptable design to BS8110:

- The profile of the unit to any profile available for manufacture
- Whether or not the unit is propped during construction of the structural screed
- The pattern of pre-stressing strands within the unit
- The position and size of the supports to the unit. Altering this may produce cantilevers.

When the design module has come up with an acceptable design (the user can modify any of the above properties in order to satisfy both the Serviceability and Ultimate Limit State conditions) it then modifies the information in the block passed to it to reflect the changes made. Once the changes have been made to the block it is written back to AutoCAD and the block's colour is changed to green symbolising the fact that the slab has been designed.

It would be unusual for an experienced designer to design each and every slab within one floor area. Depending on the area size it would not be unusual to design only one or two of the most critical slabs and assume that given the same conditions the rest of the slabs will also conform to the design criteria.

Assuming that slab No. 1-9 is the critical slab in the sample project. In design a unit size, a screed depth, whether or not the unit is propped during structural screed construction and a strand pattern for the pre-stressing strands is decided upon. Design produces calculations, modifies the block's contents and changes its colour to green. Now the layout can be looked at and a decision that slabs 1-1 through to 1-11 can be given the same properties as 1-9 is made. Since the properties of 1-9 were modified during the design process the properties of 1-1 through 1-11 need to be modified to reflect this. This is the first duty of the next toolbutton that also organises the slabs into groups with the same physical properties for production purposes.

7.9 Group Slabs

Slabs can only be grouped with a slab that has already been through the design process. The designed slab is considered the base slab for the group and it is its properties to which all other slabs are compared. The design properties of all slabs are set equal to that of the designed slab. As well as changing their design properties slabs also need to be put into production groups. Slabs that have the same unit size and strand pattern form a unique production group. Within this production group slabs will vary according to their physical properties i.e. overall span, number of holes, presence of splayed ends etc. Those with the exact same physical properties are considered to be of a particular type within a production group. These criteria provide a method for sorting all the slabs to be produced for any one contract. The sample project can be looked at to see how grouping affects its slabs. A decision made earlier established that slab 1-9 was the critical slab and it was designed. All the other slabs now need to be grouped in the drawing with respect to 1-9. On selecting the group toolbutton a prompt will be given to select a slab with which to group slabs. At this stage select a single slab that has the color acGreen. Upon selecting this slab it will be noticed that in addition to its unique identity 1-9 this slab will be given an additional identity of the form p* t* where the *

will be substituted by an integer. This identity indicates the slabs production group and also the type within the production group.

Slab 1-9 is the base slab for slabs 1-1 to 1-11. Slab 1-9 has a production group No. 1 and a type No. 1. Each of the other slabs have been given the same design properties and hence have the same production number while their type numbers vary from 1 to 9. Type number 5 is the only one that is duplicated and there are in total three of this type of slab within the production group number 1.

The user is also prompted at this stage to select all the other slabs that they wish to group with 1-9. Once the user returns their selection one by one the slabs they chose turn from their default color byBlock to the designed color of acGreen as well as assuming an additional identity denoting their production group and type within that group. Applying the group button to the sample project produces the following layout:

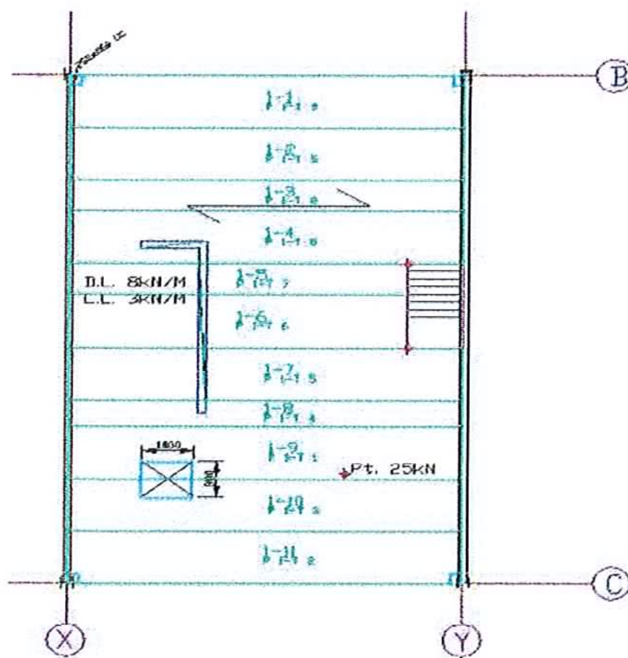


Fig 7.9.1 Grouped Slabs

Arriving at this layout requires quite a few considerations. The first thing to do after the user makes their choice for the base slab is to get its design properties:

- Unit size
- Strand pattern

- Propped value (true/false)
- Screed depth

Changes can then be made to the properties of the slabs to be grouped with this base slab. After retrieving these values from the base slab they must then be put into a production group. The production group is basically an AutoCAD block that contains a series of slabs (also AutoCAD blocks) whose unit size and strand pattern are identical. With this in mind the block should be given a name that is a combination of the slab's unit size and its strand pattern. It's strand pattern is a combination of integers which look like this:222222222200000000-25.00-52.00-115.00

The first 18 integers are strands and have values of 0 (no strand) to 3 (12.5mmØ strand), the last 3 numbers between the hyphens representing the concrete cover to the strand layers (the strand pattern string is actually used by the design module to interrogate a database containing complete strand information). The unit size refers to the manufacturers available profiles three of which were worked on:

1. prof115 (a 115mm deep unit with no hollowcore inserts)
2. prof150 (a 150mm deep unit with 4 polystyrene inserts)
3. prof200 (a 200mm deep unit with 4 polystyrene inserts)

This gives a production block with a name something similar to this;
Prof200-222222222200000000-25.00-52.00-115.00

With the base slab the drawing can be searched for a production block with the appropriate name. If the production block does not exist it needs to be created and added to the base slab as type No. 1. If the production block does exist, its elements are iterated through comparing each to the base block on a physical characteristic basis. This comparison needs to test a few different hypotheses in order to determine whether the slab is unique (and hence a new type) or not in which case the number of slabs of this type is incremented by a value of 1.

The test hypotheses for slabs to be equal within a production group are as follows:

- Their handles must not be equal (a handle is a unique identifier given to an object by AutoCAD. Slabs in this case equate to AutoCAD blocks and therefore

have handles. (I don't want to add a slab twice to a production group by mistake)

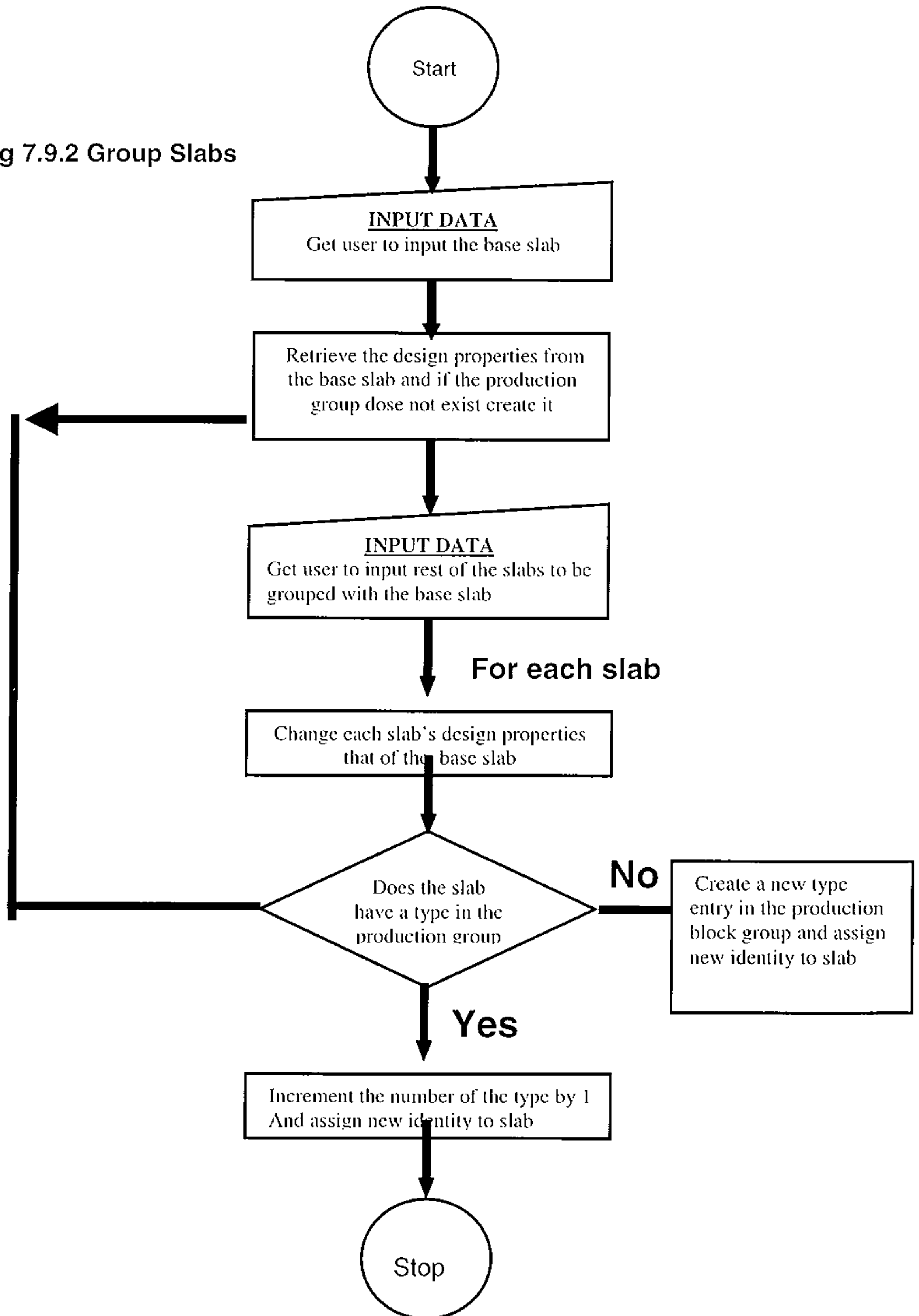
- Their radii of gyration about the x-axis must be equal (a good comparison to start with on the basis of physical properties)
- Their areas must be the same (this will filter slabs with the same radius of gyration but with different areas. Slabs of the same cross-section and different spans have the same radius of gyration)
- The position of their centroids with respect to the left hand corner of the slabs must be the same (this will filter slabs that are mirror images of themselves e.g. 1-9 and 1-10.

Only after each of these hypotheses has been passed can slabs be considered to be of the same type. Once 2 slabs are found to be of the same type, the value of the attribute that contains the total number of the slab type already contained in the production group is incremented.

At this stage a new identifier can be added to the slab indication it's production group number and its type number.

Now it is necessary to process each of the other slabs the user has selected to be grouped with the base slab. For each slab, its design properties are changed to that of the base slab (unit size, strand pattern, propped value and screed depth) and having done this each slab is added to the production group in the same manner as described for the base slab. Once the production group has been created it is simply a matter of reading the information contained within the group in order to produce production sheets. There follows a flowchart for the group toolbutton that can be implemented in VBA.

Fig 7.9.2 Group Slabs



7.10 Production Sheets

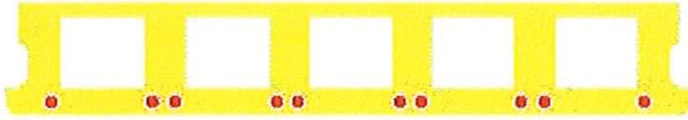
Most of the work for this toolbutton has been done when the slabs were grouped together. In doing this a production group was created (a block within the drawing) which this toolbutton is going to read. On selection of this button the user is prompted to select a slab within the drawing for which they want to create production sheets. The resulting sheets refer to a particular strand pattern and unit size. When a slab is selected on-screen it's unit size and strand pattern are read. The block corresponding to the combination of these properties (i.e. the production block for this strand pattern and unit size) is located and information pertaining to each unique slab is assimilated and sent to Microsoft Word from where the production sheets can be printed (automation of external programmes from AutoCAD, Microsoft Excel being the example was discussed earlier).
The production sheets for the sample project look like this:

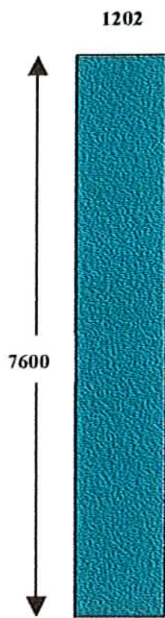
Plate 7.10.1 Production Sheet 1

Production Sheet Job ID : BuildUp Const.

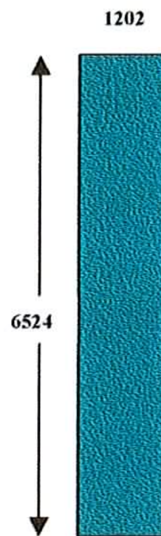
Date: 02/07/2002

200mm Slab

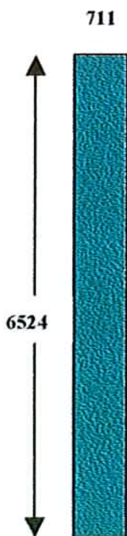




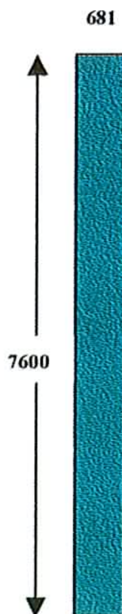
P 1-T 5
TOTAL No : 3



P 1-T 6
TOTAL No : 1



P 1-T 7
TOTAL No : 1



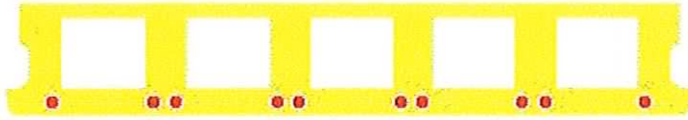
P 1-T 8
TOTAL No : 1

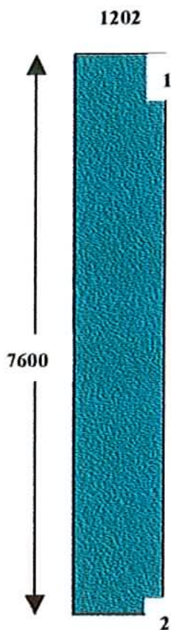
Plate 7.10.2 Production Sheet 2

Production Sheet Job ID : BuildUp Const.

Date: 02/07/2002

200mm Slab





P 1-T 9

TOTAL No : 1

Holes:

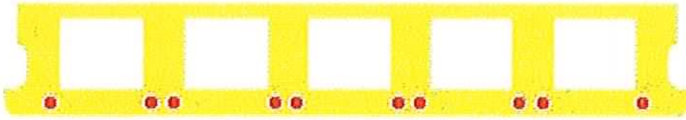
size	x,y
230 x 230	972 , 0
230 x 230	972 , 7370

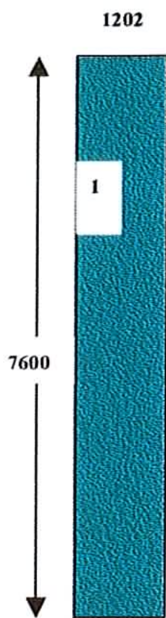
Plate 7.10.3 Production Sheet 3

Production Sheet Job ID : BuildUp Const.

Date: 02/07/2002

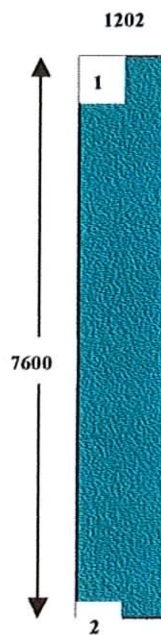
200mm Slab





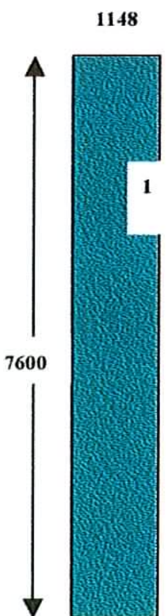
P 1-T 1
TOTAL No : 1
Holes:

size	x,y
400 x 1000	0 , 1423



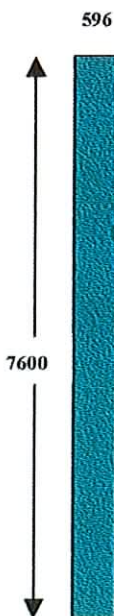
P 1-T 2
TOTAL No : 1
Holes:

size	x,y
230 x 230	0 , 0
230 x 230	0 , 7370



P 1-T 3
TOTAL No : 1
Holes:

size	x,y
400 x 1000	748 , 1423



P 1-T 4
TOTAL No : 1

7.1 Toggle Layers

This is basically a utility that can be used at any time. It is useful for drawings that have a huge amount of detail. This button will turn off all layers except the "SlabOutline" layer to which all of the preceding toolbuttons draw.

7.12 Reset

Again this is a utility button. If the user chooses the Esc key during the operation of one of the earlier toolbuttons it may leave the drawing unusable to subsequent toolbutton actions (blocks may exist when they are not supposed to etc). This toolbutton erases all blocks and their contents that may have been created by the toolbar in the first place.

7.13 Concluding AutoCAD

This concludes the development within AutoCAD. In the earlier part of this section it was attempted to give the user a good insight into programming with VBA in AutoCAD. Although very powerful VBA is not equipped to do everything and depending on the complexity of the programme it might be necessary to step outside the VBA IDE every now and again. For this purpose Delphi was found to be a very powerful programming tool indeed.

Chapter 8 Delphi

8.1 Where To Begin?

For now the answer to this question is not at the beginning! Delphi is simply too expansive and the learning curve too steep to expect to bring anyone to a competent level of programming before looking at the DLLs that were created for AutoCAD. As mentioned before Delphi is written in Object Pascal and is a hierarchical environment based on a class structure the top-level object being TObject.

Delphi was used to create an Automation Controller for AutoCAD. These controllers exist thanks to The Component Object Model or COM.

8.2 COM Revisited

COM is a component software architecture that allows applications and systems to be built from components supplied by different software vendors. COM is the underlying architecture that forms the foundation for higher-level software services, like those provided by ActiveX. ActiveX services span various aspects of component software, including custom controls, data transfer, and other software interactions. These services provide distinctly different functionality to the user; however, they share a fundamental requirement for a mechanism that allows binary software components, supplied by different software vendors, to connect to and communicate with each other in a well-defined manner. This mechanism is supplied by the COM component software architecture in a way such that it:

- Defines a binary standard for component interoperability
- Is programming language-independent
- Is provided on multiple platforms (Microsoft Windows, Microsoft Windows NT, Apple Macintosh, UNIX)
- Provides for robust evolution of component-based applications and systems
- Is extensible

- Communicates between components, even across process and network boundaries
- Shares memory management between components
- Reports errors and status
- Loads components dynamically

The most fundamental question COM addresses is: How can a system be designed such that binary executables from different vendors, written in different parts of the world, and at different times are able to interoperate? To solve this problem, solutions to four specific problems had to be found:

- Basic interoperability—How can developers create their own unique components, yet be assured that these components will interoperate with other components built by different developers?
- Versioning—How can one system component be upgraded without requiring all the system components to be upgraded?
- Language independence—How can components written in different languages communicate?
- Transparent cross-process interoperability—How can developers be given the flexibility to write components to run in process or cross-process (and eventually cross-network), using one simple programming model?

Additionally, high performance is a requirement for component software architecture. While cross-process and cross-network transparency is commendable, it is critical for the commercial success of a binary component marketplace that components interacting within the same address space be able to utilize each other's services without any undue system overhead. Otherwise, the components will not realistically be scalable down to very small, lightweight pieces of software equivalent to Delphi classes or graphical user-interface (GUI) controls.

8.3 The Basics Of COM

COM has several building blocks that include:

- A binary standard for function calling between components.

- A provision for strongly typed groupings of functions into interfaces.
- A base interface providing:
- A way for components to dynamically discover the interfaces implemented by other components.
- Reference counting to allow components to track their own lifetime and delete themselves when appropriate.
- A mechanism to uniquely identify components and their interfaces.
- A component loader to set up component interactions and additionally in the cross-process and cross-network cases to help manage component interactions.

8.4 Virtual Method Tables

For any given platform (hardware and operating system combination), COM defines a standard way to lay out virtual function tables (vtables) in memory, and a standard way to call functions through the vtables.

In Delphi a class-type value is stored as a 32-bit pointer to an instance of the class in memory, which is called an object. The internal data format of an object resembles that of a record. The object's fields are stored in order of declaration as a sequence of contiguous variables. Fields are always aligned, corresponding to an unpacked record type. Any fields inherited from an ancestor class are stored before the new fields defined in the descendant class.

The first 4-byte field of every object is a pointer to the virtual method table (VMT) of the class. There is exactly one VMT per class (not one per object); distinct class types, no matter how similar, never share a VMT. VMTs are built automatically by the compiler, and are never directly manipulated by a program.

The layout of a VMT is compatible with COM. At positive offsets, a VMT consists of a list of 32-bit method pointers—one per user-defined virtual method in the class type—in order of declaration. Each slot contains the address of the corresponding virtual method's entry point. At negative offsets, a VMT contains a number of fields that are internal to Object Pascal's implementation.

8.5 Interfaces

In COM, applications interact with each other and with the system through collections of functions called interfaces. A COM interface is a strongly typed contract between software components to provide a small but useful set of semantically related operations (methods). An interface is the definition of an expected behavior and expected responsibilities. Interface names begin with "I" by convention.

Given that an interface is a contractual way for a component object to expose its services, there are four very important points to understand:

- An interface is not a class. While a class can be instantiated to form a component object, an interface cannot be instantiated by itself because it carries no implementation. A component object must implement that interface and that component object must be instantiated for there to be an interface.
- An interface is not a component object. An interface is just a related group of functions and is the binary standard through which clients and component objects communicate. The component object can be implemented in any language with any internal state representation, so long as it can provide pointers to interface member functions.
- Clients only interact with pointers to interfaces. When a client has access to a component object, it has nothing more than a pointer through which it can access the functions in the interface, called simply an interface pointer. The pointer is opaque; it hides all aspects of internal implementation. The component object's data cannot be seen, as opposed to C++ object pointers through which a client may directly access the object's data. In COM, the client can call only methods of the interface to which it has a pointer. This encapsulation is what allows COM to provide the efficient binary standard that enables local/remote transparency.
- Component objects can implement multiple interfaces. A component object can—and typically does—implement more than one interface. That is, the class has more than one set of services to provide. For example, a class might

support the ability to exchange data with clients as well as the ability to save its persistent state information (the data it would need to reload to return to its current state) into a file at the client's request. Each of these abilities is expressed through a different interface (IDataObject and IPersistFile), so the component object must implement two interfaces.

- Interfaces are strongly typed. Every interface has its own interface identifier, a globally unique ID (GUID: 128-bit integers that are guaranteed to be unique in the world across space and time) thereby eliminating any chance of collision that would occur with user-defined names. The difference between components and interfaces has two important implications. If a developer creates a new interface, they must also create a new identifier for that interface. When a developer uses an interface, they must use the identifier for the interface to request a pointer to the interface. This explicit identification improves robustness by eliminating naming conflicts that would result in run-time failure.
- Interfaces are immutable. COM interfaces are never versioned, which means that version conflicts between new and old components are avoided. A new version of an interface, created by adding more functions or changing semantics, is an entirely new interface and is assigned a new unique identifier. Therefore, a new interface does not conflict with an old interface even if all that changed is one operation or semantics (but not even the syntax) of an existing method. Note that as an implementation matter, it is likely that two very similar interfaces can share a common internal implementation. For example, if a new interface adds only one method to an existing interface, and the component author wishes to support both old-style and new style clients, they would express both collections of capabilities through two interfaces, but internally implement the old interfaces as a proper subset of the implementation of the new.

COM defines one special interface, **IUnknown,** to implement some essential functionality. All component objects are required to implement the **IUnknown** interface, and conveniently, all other COM interfaces derive from **IUnknown.**

IUnknown has three methods: ***QueryInterface, AddRef, and Release***. In Delphi, the declaration of IUnknown looks like this:

```
IUnknown = interface
  ['{00000000-0000-0000-C000-000000000046}']
  function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
  function _AddRef: Integer; stdcall;
  function _Release: Integer; stdcall;
 end;
```

AddRef and Release are simple reference counting methods. A component object's ***AddRef*** method is called when another component object is using the interface; the component object's Release method is called when the other component no longer requires use of that interface. While the component object's reference count is nonzero, it must remain in memory; when the reference count becomes zero, the component object can safely unload itself because no other components hold references to it.

QueryInterface is the mechanism that allows clients to dynamically discover (at run time) whether or not an interface is supported by a component object; at the same time, it is the mechanism that a client uses to get an interface pointer from a component object. When an application wants to use some function of a component object, it calls that object's ***QueryInterface***, requesting a pointer to the interface that implements the desired function. If the component object supports that interface, it will return the appropriate interface pointer and a success code. If the component object doesn't support the requested interface, then it will return an error value. The application will then examine the return code; if successful, it will use the interface pointer to access the desired method. If the ***QueryInterface*** failed, the application will take some other action, letting the user know that the desired method is not available.

In COM, an object is some piece of compiled code that provides some service to the rest of the system. To avoid confusion, it is probably best to refer to a COM object as a "component object" or simply a "component." Component objects support a base interface called **IUnknown**, along with any combination of other

interfaces, depending on what functionality the component object chooses to expose.

Component objects usually have some associated data and always access other component objects through interface pointers. This is a primary architectural feature of COM, because it allows COM to completely preserve encapsulation of data and processing, a fundamental requirement of a true component software standard.

An object whose class implements the **IDispatch** interface (declared in the System unit) is an Automation object.

Dispatch interface types define the methods and properties that an Automation object implements through **IDispatch**. Calls to methods of a dispatch interface are routed through IDispatch's ***Invoke*** method at runtime.

8.6 COM and the Client Server Model

The interaction between component objects and the users of those component objects in COM is in one sense based on a client/server model. The term "client" could be referred to as some piece of code that is using the services of a component object. Because a component object supplies services, the implementation of that component is usually called the "server"—the component object that serves those capabilities. A client/server architecture in any computing environment leads to greater robustness: If a server process crashes or is otherwise disconnected from a client, the client can handle that problem gracefully and even restart the server if necessary. As robustness is a primary goal in COM, a client/server model naturally fits. Because COM allows clients and servers to exist in different process spaces (as desired by component providers), crash protection can be provided between the different components making up an application. For example, if one component in a component-based application fails, the entire application will not crash. In contrast, object models that are only in process cannot provide this same fault tolerance. The ability to cleanly separate object clients and object servers in different process spaces is very important for a

component software standard that promises to support sophisticated applications. COM is unique in allowing clients to also represent themselves as servers.

8.7 Servers: In-Process and Out-of-Process

In general a "server" is some piece of code that implements some component object such that the Component Object Library and its services can run that code and have it create component objects.

Any specific server can be implemented in one of a number of flavors depending on the structure of the code module and its relationship to the client process that will be using it. A server is either "in-process," which means its code executes in the same process space as the client (as a DLL), or "out-of-process", which means it runs in another process on the same machine or in another process on a remote machine (as a .EXE file). These three types of servers are called "in-process", "local", and "remote."

Component object implementers choose the type of server based on the requirements of implementation and deployment. COM is designed to handle all situations from those that require the deployment of many small, lightweight in-process components (like OLE Controls, but conceivably even smaller) up to those that require deployment of huge components, such as a central corporate database server. All component objects look the same to client applications, whether they are in process, local, or remote.

Chapter 9 Dynamic Link Libraries

9.1 Type Libraries

A type library (.tlb) is a binary file that stores information about a COM object's properties and methods in a form that is accessible to other applications at run time. Using a type library, an application or browser can determine which interfaces an object supports, and invoke an object's interface methods. This can occur even if the object and client applications were written in different programming languages.

Delphi records the description of the ActiveX server's interfaces in a type library that it automatically creates when creating the ActiveX object. Type libraries do not, however, store the actual objects described—they store only information about objects. (They might also contain immediate data such as constant values.)
At this stage the theory behind the programming that goes into making up a custom ActiveX DLL has been discussed. This is now used in order to create two ActiveX DLLs for the AutoCAD programme. These ActiveX objects look after two stages in the manufacturing process, the drawing stage and the design stage.

9.2 Making A Start: The Simplest Of ActiveX DLLs

Both DLLs use Delphi forms with which the user can interact in order to achieve the desired results. First, creating a Delphi form in AutoCAD from a Delphi compiled DLL, was examined. This form had one button which when selected showed another form with a close button. It basically does nothing but once achieved the form can then be designed to do anything.

From the file menu New is chosen:

Delphi displays the new items dialogue box.

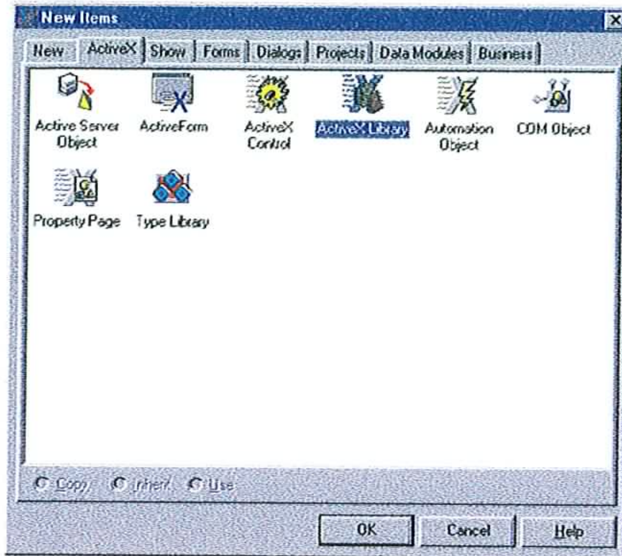


Fig 9.2.1 Adding an ActiveX Library

At this stage it is necessary to add the forms to the project that AutoCAD is going to show.

Choose File-New-Form

Delphi adds a new form to the project and it's associated unit. Onto this form drag two buttons from the standard palette to which some new code can be added.

Since this is going to be the first form to be shown by AutoCAD it was saved as Shown_First.pas in the same directory as saved in the project.

Choose File-New-Form

Delphi adds another form to the project. A close button can then be dropped onto this form. Since this form is going to be created by the first form it was as Secondary_Form.pas.

Now some code can be added to the buttons on the Shown_First form.

One of them simply closes the project the other shows the secondary form.

The code looks like this:

```
procedure TFirst.Show_FormClick(Sender: TObject);
var
second:TSecond;
```

```
begin
   Second:=TSecond.Create(self);
    try
      Second.showmodal;
   finally
      Second.free;
   end;
end;

procedure TFirst.CloseClick(Sender: TObject);
begin
  Close;
end;
```

The forms look like these.

Fig 9.2.2 Both member forms of the ActiveX dll

In order to make these forms behave like any other dialogue boxes in AutoCAD (when AutoCAD is minimised the forms will also minimise) it is necessary to make AutoCAD the owner of the forms i.e. the AutoCAD window is the parent window of the forms. This is done by creating parameters for the first form (since it owns the second form by default AutoCAD will also own the second form).

The following code is needed to make AutoCAD the owner of the first form:

```
type
 TFirst = class(TForm)
   Shown_First: TButton;
   Close: TButton;
   procedure Shown_FirstClick(Sender: TObject);
   procedure CloseClick(Sender: TObject);
 private
```

```
    { Private declarations }
    FAcadWindow: HWND;
    Function GetAcadWindow: HWnd;
  protected
    procedure CreateParams(var Params: TCreateParams); override;
public
    { Public declarations }
    Property AcadWindow: HWND read FAcadWindow;
end;
Implementation
procedure TFirst.CreateParams(var Params: TCreateParams);
begin
  inherited CreateParams(Params);
  FAcadWindow := GetAcadWindow;
   If IsWindow(FAcadWindow) then
    Params.WndParent := FAcadWindow;
end;
Function TFirst.GetAcadWindow: HWND;
var
  Func: Function: HWnd; cdecl;
begin
  Result := HWND(0);
  Func:= GetProcAddress(GetModuleHandle('ACAD.EXE'), 'adsw_acadMainWnd');
  If not Assigned(Func) then Exit;
  Result := Func;
end;
```

The ActiveX object which implements the previous code written can now be created:

On choosing File-New-ActiveX-Automation Object

Delphi displays the Automation Object Wizard

Fig 9.2.3 Automation Object Wizard

Into the CoClassName box Forms is typed. This is a **CoClass CoForms** which provides a Create and ***CreateRemote*** method to create instances of the default interface **IForms** exposed by the **CoClass**. The functions are intended to be used by clients wishing to automate the **CoClass** objects exposed by the server.

On selecting OK Delphi does a number of things:

- Creates two Interfaces **IForms** and **IformsDisp**
- Creates a Class **CoForms** used to create instances of the default interface **IForms** exposed by the **CoClass**
- Creates a Class **TForms** which is implemented by **IForms**
- Creates a Type Library to describe the methods and properties of the ActiveX object
- Creates a unit where the methods and properties of the interface **Iforms** can be implemented
- Creates a Class **TFormsProperties** which acts as the server for the class **TForms**

Delphi then shows its type library editor:

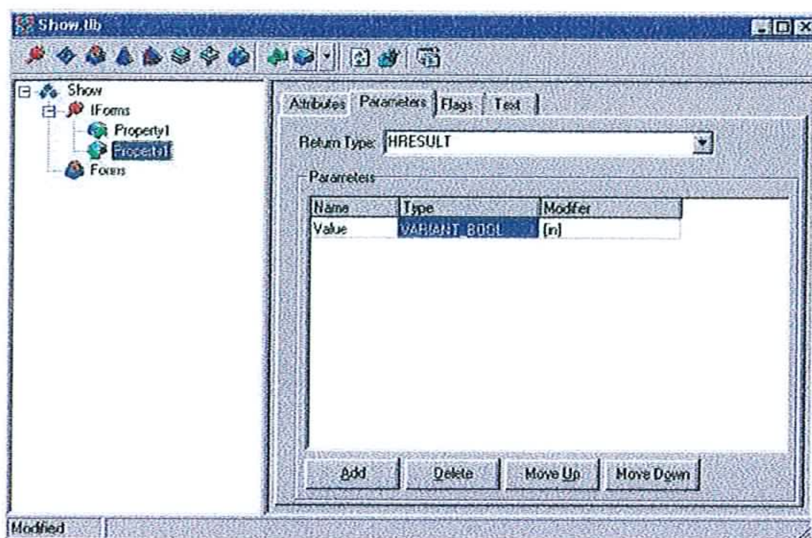


Fig 9.2.4 Delphi Type Library Editor

The interface **Iforms** was highlighted **and** the new property button from the toolbar was selected. The new property added was called Visible. On the parameters tab in the drop down list for Type in the parameters window VariantBool was chosen as the type for parameter. On choosing refresh implementation Delphi created the new properties in the new unit. The unit was saved as Forms_IMPL

The code for the Visible property is very simple and once the ***Initialise and Destroy*** procedures have been added the project is complete. The code in Forms_IMPL then looks like this:

```
unit Forms_IMPL;interface
uses
 ComObj, ActiveX, Windows, AcForm_TLB, StdVcl, First;
type
 TForms = class(TAutoObject, IForms)
 private
  FFirst: TFirst;
 protected
  function Get_Visible: WordBool; safecall;
  procedure Set_Visible(Value: WordBool); safecall;
  { Protected declarations }
 public
  Procedure Initialize; override;
  Destructor Destroy; override;
```

```
 end;

implementation
uses ComServ;

procedure TForms.Initialize;
begin
 Inherited Initialize;
 FFirst := TFirst.Create(nil);
end;

destructor TForms.Destroy;
begin
 FFirst.Free;
 inherited destroy;
end;

function TForms.Get_Visible: WordBool;
begin
 Result := FFirst.Visible;
end;

procedure TForms.Set_Visible(Value: WordBool);
begin
 FFirst.Visible := Value;
end;

initialization
 TAutoObjectFactory.Create(ComServer, TForms, Class_Forms,
   ciMultiInstance, tmApartment);
end.
```

Now before the ActiveX server can be used it must first be registered on the system. If from the run menu 'Register ActiveX Server' is chosen Delphi makes the appropriate entries of GUIDs etc to the system registry. The server is now available to AutoCAD and a small bit of VBA code is necessary to access it. The first thing to

be done from the VBA IDE is to set a reference to the new DLL (by browsing to where DLL was saved). Once this reference has been set, the library Show appears in the object browser window. It shows a class called Forms that has a single visible property. To show the Delphi form from AutoCAD add this code to the VBA project:

```
Dim showfrms As Show.Forms 'this must be a module level declaration
Public Sub ShowForms()
Set showfrms = New Show.Forms
showfrms.Visible = True
End Sub
```

There were two Delphi DLLs created for AutoCAD :

1. The draw DLL which divides up the floor area into its separate slabs
2. The design DLL which produces a set of design calculations for a particular slab

9.3 The Draw DLL

This DLL deals with rendering the floor area into a series of slab objects that can then be manipulated by both the design DLL and the production sheets toolbutton. It is simply a Delphi form that responds to actions carried out by the user. There are three main events to which the form responds:

1. Mouse Down Event (equivalent to depressing the left mouse button)
2. Mouse Move Event
3. Mouse Up Event (equivalent to releasing the left mouse button)

When an acceptable layout is achieved selecting the OK button writes back the layout to AutoCAD (Note: to access AutoCAD objects from Delphi, first import its type library to the project where Delphi will create the unit AutoCAD_TLB.pas). The bulk of the Draw DLL is taken up by a picture control onto which all drawing is done. The defaults of this picture are set up to look like the AutoCAD environment so that the transition from AutoCAD to the DLL is seamless to the user. When this DLL is called from AutoCAD an initial method called ***CreateInitialSlicing*** is executed. This method takes a parameter, **IDispatch,** which it gets from the

AutoCAD VBA routine that calls it. The **IDispatch** parameter is a pointer to the current drawing in AutoCAD.

On connecting to the AutoCAD drawing the DLL looks for the individual blocks containing the information relative to the floor area to be slabbed. As mentioned before these blocks are:

1. Floor Outline
2. Hole Details
3. Wall Details

The ***CreateInitialSlicing*** method then extracts the coordinates of the floor outline from the appropriate block and draws it to the bitmap on the form. As well as this it also creates an array of slices for each individual side of the floor outline that are parallel to the side. Having completed these tasks it then shows the form to the user. No slicing arrangement is apparent until the user moves the mouse cursor inside the floor area. Once inside the floor area depending upon which side the cursor is closest to a slicing arrangement is presented to the user a view of which is shown below:

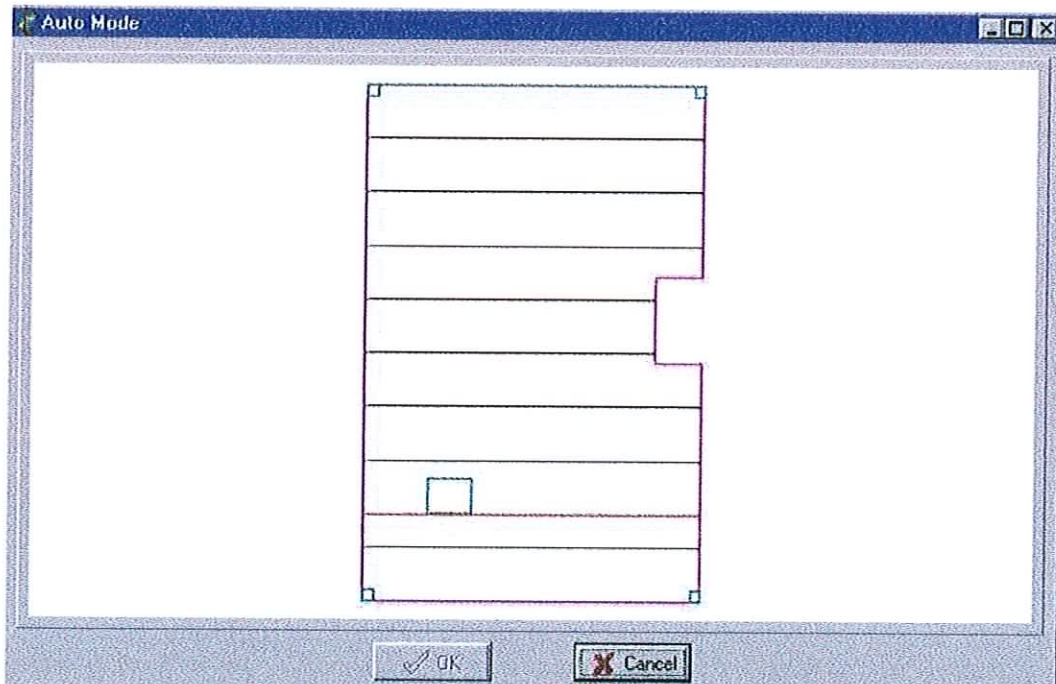


Fig 9.3.1 The Draw Editor

At this stage the user has 2 choices, either right clicking on the mouse that will fix the arrangement as presented or left clicking the mouse to slice individual areas within the floor area. On right clicking the only thing that can be altered in the arrangement is the width of the last but one slice (it is industry practice to make this the narrow slice). By placing the mouse on the second last slice (the red one) and left clicking the slice can be dragged to alter the width of the last but one slab. Its width can be adjusted to a maximum of 1200mm consequently narrowing the previous slab. The experienced designer should have an intuition as to the optimum width of this slab depending on the presented layout.

For the case above this option would probably not be relevant. To achieve the slicing arrangement aground the stair opening it is necessary to slab to the corner of the stairs, provide two slabs covering the width of the stair opening, and then slab the rest of the floor area. Upon left clicking the mouse the user is allowed to slice individual areas within the floor area. With this tool it is possible to drag a slab edge to the corner of the stair opening. On releasing the mouse button in the vicinity of the corner of the stair opening slabs are drawn from this point to the edge of the floor outline. Again a narrow slab will be encountered.

In the following case instead of right clicking the user has chosen the left click option to slice from the top edge to the corner of the stair opening. Slicing proceeds from this corner to the next corner of the stair opening. Note the slab width labels in the bottom left hand corner. These labels appear when the user opts to alter the width of the last but one slice. The labels indicate the width of both slabs dynamically.

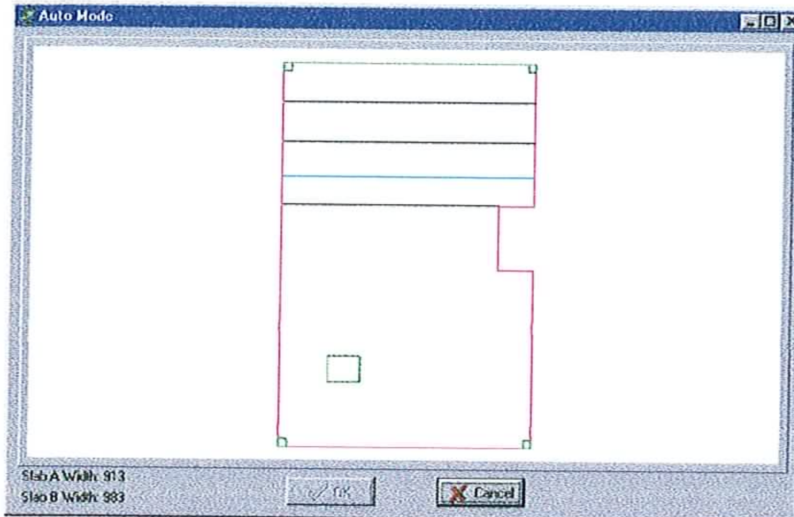


Fig 9.3.2 Dynamic Slicing

The user continues to slab to the opposite corner of the stair opening. Now at this stage an experienced designer would know that the large opening is too big for a single slab to contain. In practice the adjacent slabs would share the entire hole. To automate this process the user is given the option of releasing the left mouse button inside the hole. The process detects the hole and a slab edge is created at the centre of the hole from which slabs are generated. The user can now complete their layout by dragging the mouse from the last edge to the lower corner of the floor area to complete a valid layout (see the corresponding toolbutton in AutoCAD). At this stage the Ok button is enabled. Selecting it accepts the layout and commences building up the information for individual slabs. Algebraically a "Slab Object" is constructed which is used to build AutoCAD Blocks for writing to the drawing.

This " Slab Object" type is defined in the code in the following way:

```
TComplete_Slab = Record
  outline:TSlab;
  holes:THole;
  walls:TWalls;
  points:TPointLds;
end;
```

```
TCoord=record
  x,y:double;
end;
TLine = Array[0..1] of TCoord;
TSide = Array of TLine;
TSlab = Array of TSide;
Thole = Array of TSide;
TWallRecord=record
  Wall:TLine;
  deadload:widestring;
  liveload:widestring;
end;
TPointLoad=record
  PtLoad:TCoord;
  PtDL:Widestring;
  PtLL:Widestring;
end;
```

The **TComplete_Slab** record is built up from intersection points calculated from the layout and using the information contained in the drawing about walls and point loads. Once this record has been built it is a straightforward matter to convert all the information into an AutoCAD block that can be written back to the drawing. The structure of this block is described in detail in the Get_Slab_Boundary toolbutton description. Values for attributes that cannot be ascertained at this time are given defaults to be changed at a later date. One by one the information is built up for each slab and one by one they are written back to the AutoCAD drawing. When the DLL has completed writing all the slabs back to AutoCAD from the accepted layout devised on the Delphi form, the form disappears and the user is back in the AutoCAD environment. The appropriate slabs including reference numbers have been drawn in the AutoCAD drawing. These slabs can be simply selected on-screen by selecting any element which is owned by the block/slab such as a wall, hole, point load, slab identity etc. Since the slab is defined as an AutoCAD block selecting a single element selects the whole block (which is actually an instance of

a block insertion i.e. a block reference in VBA) that is useful when it comes to performing operations on individual slabs (design calculations etc.).
Provided there are no slabs hanging from the slab for which a design is required it can be selected on-screen and sent to the design DLL (If there are hanging slabs they must be dealt with first with the AutoCAD hangers toolbutton).

9.4 The Design Dll

A reference to the slab that is to be designed is sent to the Design DLL in the form of **IDispatch** (a pointer to the memory address containing the block information for the slab). The first thing the Design DLL does is to deconstruct the AutoCAD block that is referenced by **IDispatch** into the component parts needed to perform an initial design. The information needed for this initial design is as follows:

- The overall span of the slab
- The width of the supports
- The position of the supports.
- Whether or not to prop the slab
- The applied Dulls
- The depth of the screed
- The positions and sizes if any holes exist
- The location of any walls if they exist and there associated dead and live loads
- The location and value of any point loads if they exist
- The strand pattern for the slab

All of this information is extracted from the AutoCAD block the reference of which was given by the user when they selected a slab on-screen.
The design dll is comprised of a series of user interactive forms that can change the design properties of the design slab in order to satisfy design criteria laid down in BS8110. Given the initial properties sent over from AutoCAD two calculations are performed and a results sheet is shown to the user in which they can check:

1. Serviceability Limit State Calculations
2. Ultimate Limit State Calculations

For SLS calculations the slab is considered at two stages of its lifetime i.e. at transfer (the stress induced in the pre-stressing strands is transferred to the concrete which at this stage of its life would have a transfer strength of $30N/mm^2$) and at service (the concrete has a service strength of $50N/mm^2$ and imposed loads due to screed construction and floor area usage are considered).

The effective span of the slab is divided into 100 intervals. At each interval section properties and the corresponding stress in the top and bottom fibers of the concrete is calculated. These values are then plotted to a chart that has stress on the y-axis and distance along the span on the x-axis. Stress limits are also drawn on this chart ($14N/mm^2$ in compression and $-2.66N/mm^2$ in tension at transfer and $16N/mm^2$ in compression and $-3.18N/mm^2$ at service). The slab is deemed to have passed the SLS calculations if the stress profiles do not exceed the limits laid down.

For the ULS calculations check applied shear force against shear capacity and the applied moment against the moment capacity. Both cracked and un-cracked shear capacities are calculated at each interval along the span of the slab. A shear envelope can then be drawn representing the shear capacity of the slab along its entire span. The applied shear is then plotted along the same axes. Once the shear force diagram does not intersect the shear envelope the ULS Shear calculations have been satisfied.

Similar calculations are performed for the moment capacity. It is calculated at the same intervals, as was the shear. A moment capacity envelope is then drawn which encompasses the entire span of the slab. Applied moment is calculated at each interval. Once the applied moment does not intersect the moment envelope the ULS Moment calculations have been satisfied.

Slabs are assumed to carry 100% of applied wall and point loads although in practice screeds and key joints tend to distribute some of these loads to adjacent slabs.

A series of bitmaps are generated which when grouped together go to make up one complete set of results. These bitmaps are then pasted onto a Delphi Form which is activated by the Design DLL and presented to the user as the active on-

screen window for evaluation. The first page of the results sheet show all the physical characteristics of the slab to be designed including any walls holes or point loads to be considered. On the second page the user can view the SLS calculations and on the final page ULS calculations are displayed. The user can then decide on the basis of these calculations whether or not to change the design properties of the slab.

At any stage the user can send the calculations to a Microsoft Word document whereupon the bitmaps are pasted into a Word document that is activated by the Design DLL. When viewed in Word the design calculations look like this:

Plate 9.4.2 Results Page No.1

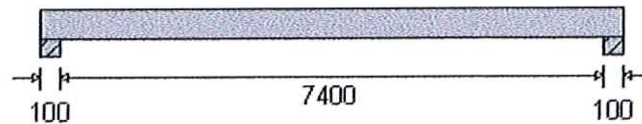
Client: **BuildUp Const.**	Date: 13-11-00	JOB REF: 1-4

Basic Design Data

Self Weight(unjointed)	4.66 kN/m²	Dead Load Factor(ULS)	1.4
Self Weight(jointed)	4.82 kN/m²	Live Load Factor(ULS)	1.6
Screed Strength	35 N/mm²	Ixx:	661.2E6
Slab Strength @ Transfer	35 N/mm²	Ybtm:	93.44 m
Slab Strength @ Service	50 N/mm²	I Comp:	1.8E9 m
Creep Factor	2.0	Y Comp:	144.52

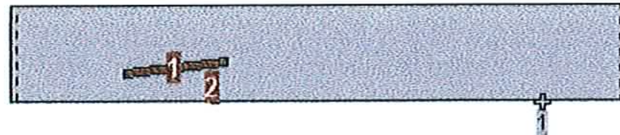
Slab Details (200mm Slab)

Legend
▲ 9.3 mm strand



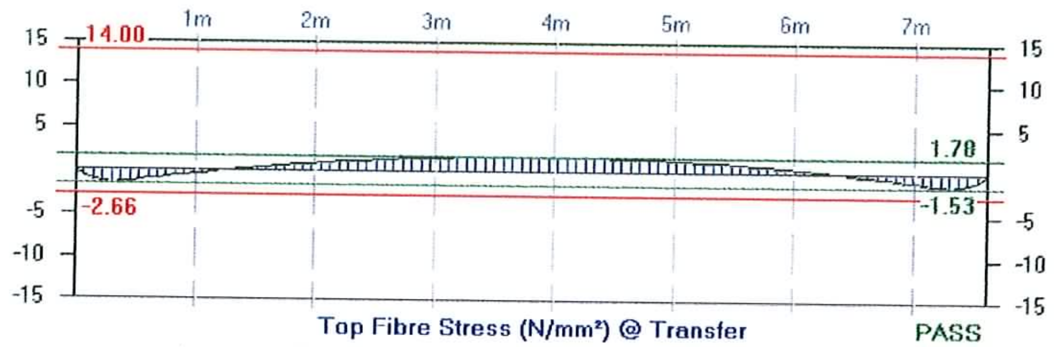
Applied Loads

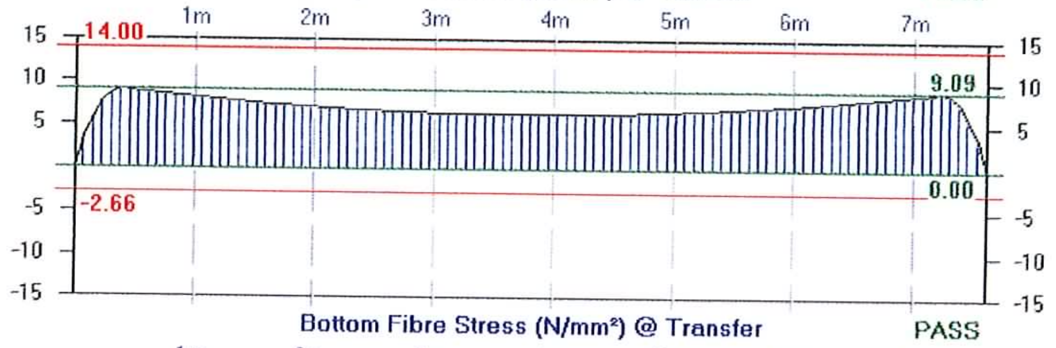
UDL No: (kN/m²)	No: 1	
Dead Load:	0.5	
Live Load:	1.5	
Point Load No: (kN)	**No: 1**	
Dead Load:	19.7	
Live Load:	5.8	
Top (mm):	1202	
Left (mm):	6524	
Wall No: (kN/m)	**No: 1**	**No: 2**
Dead Load:	8.0	8.0
Live Load:	3.0	3.0

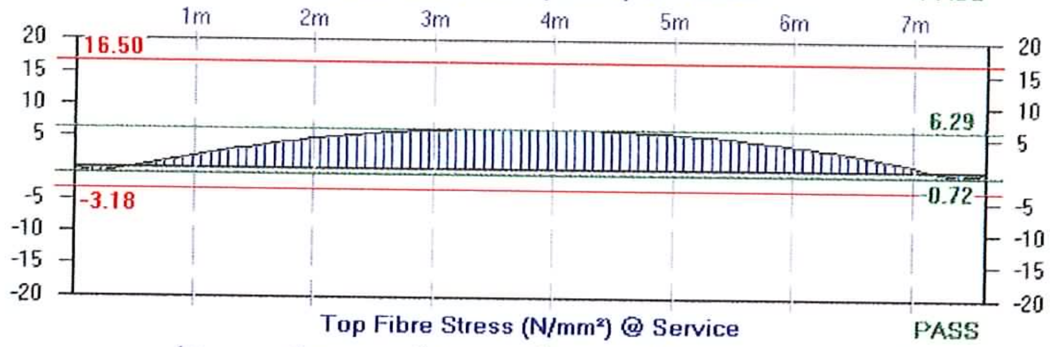


Openings

There are no openings defined.


14.00
1m
2m
3m
4m
5m
6m
7m
1.78
-2.66
-1.53
Top Fibre Stress (N/mm²) @ Transfer
PASS


14.00
1m
2m
3m
4m
5m
6m
7m
9.09
-2.66
0.00
Bottom Fibre Stress (N/mm²) @ Transfer
PASS


16.50
1m
2m
3m
4m
5m
6m
7m
6.29
-3.18
-0.72
Top Fibre Stress (N/mm²) @ Service
PASS

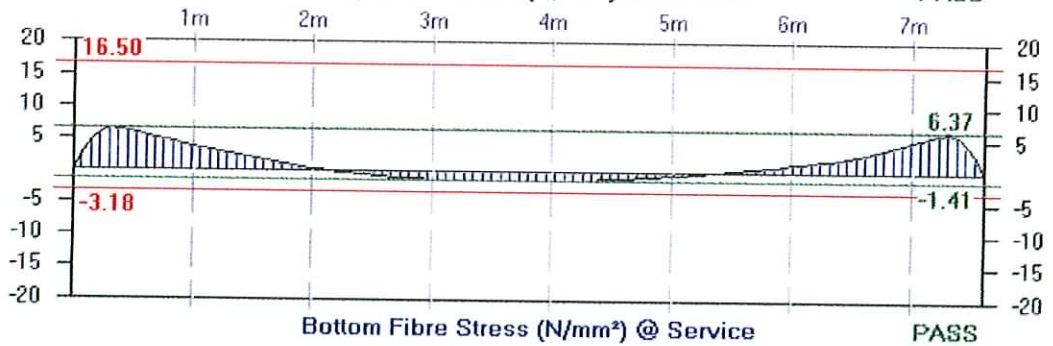
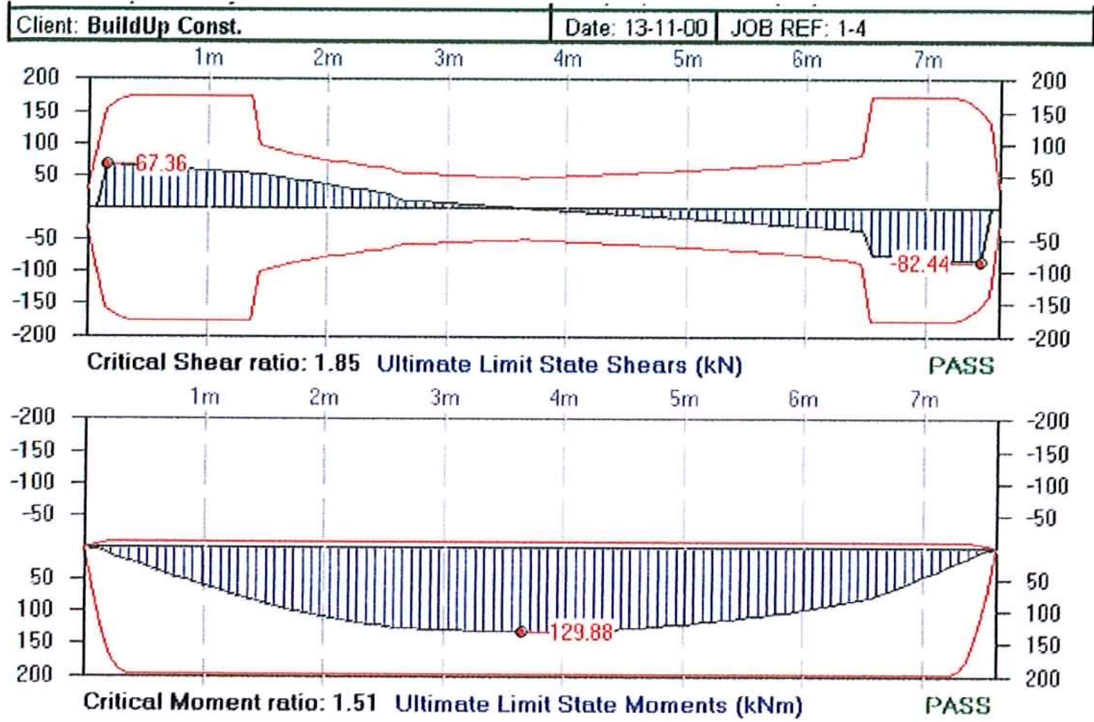

16.50
1m
2m
3m
4m
5m
6m
7m
6.37
-3.18
-1.41
Bottom Fibre Stress (N/mm²) @ Service
PASS

Plate 10.4.3 Results Page No.3



While being presented with the calculations the user is also presented with a toolbar on the Delphi form with which they can change the design parameters for the slab. The toolbar looks like this

Fig 9.4.1 Results Sheet Toolbar

The back and forward buttons simply navigate through the results pages from one to three. The Send To button simply sends the results sheets to a Microsoft Word document from where the results can be printed. The print button sends the results directly to the printer, which leaves three toolbuttons that directly affect the design of the slab.

The profile toolbutton activates a dialogue box that allows the user to change the profile of a slab to another profile that is supported.

Here the unit's size can be changed to an alternative supported size. Any number of different profiles can be supported. Also a decision whether or not to prop the construction of the structural screed with the "Propped?" check box is made on this form. Selecting OK recalculates and presents results to the user.

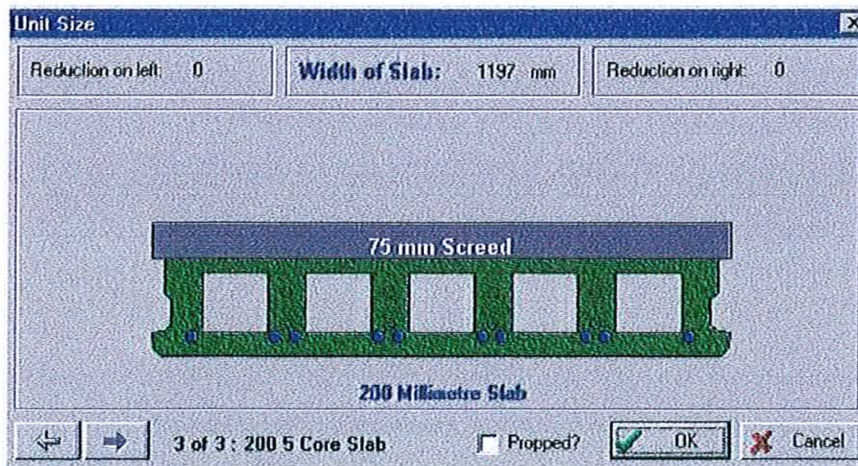


Fig 9.4.2 Changing the unit size

The strands toolbutton brings up a dialogue box in which the strand pattern of the slab can be altered.

Here the user configures the strand pattern for the given slab. Strands can be turned on and off-strand diameters can be changed. The strands lie in predetermined layers the cover of which can be changed (in industry these layers will be fixed by drilling the end-blocks of the profile mould). Again selecting OK recalculates and presents results to the user.

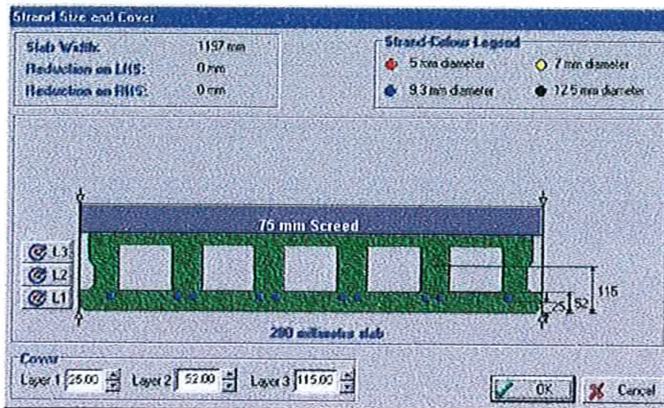


Fig 9.4.3 Changing the strand pattern

The final design parameters that are dynamic are presented by the spans toolbutton:

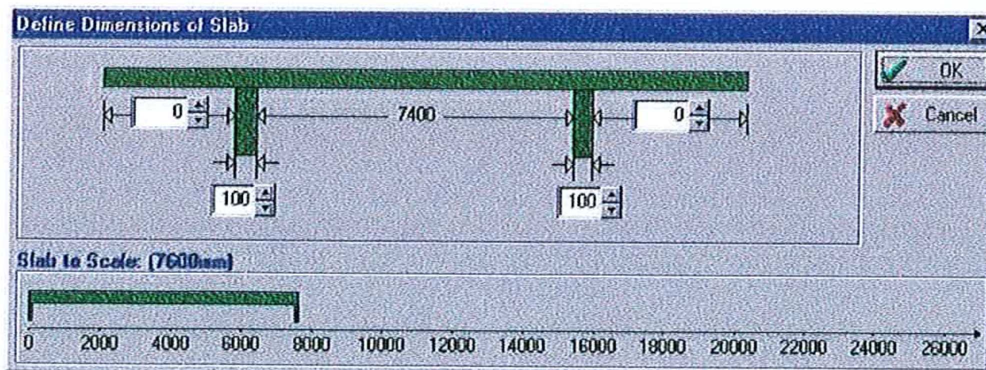


Fig 9.4.4 Creating cantilevers

The location and the width of the supports can be changed. Consequently cantilevers can be set up. Selecting OK for a configuration recalculates and presents results for examination.
With the use of the toolbuttons described above it is possible to test the design of many slabs with the desire to satisfy the physical conditions of its use. Once an acceptable design combination has been achieved the user selects OK on the main form. Each design parameter that came from AutoCAD in the block is checked for a change and if one is applicable the appropriate change is written to the block. Finally the block's colour is changed to green to graphically indicate that it has been designed and the Design DLL is released to await another call!

Chapter 10 Conclusions

10.1 Achieved Developments

While developing this IT package the AutoCAD environment proved to be an extremely user-friendly versatile product to work with. Only a small subset of its vast drawing capabilities was used. The nature of the drawing objects provided by AutoCAD (such as the AcadLine and AcadRegion), with their methods and properties, avoided the need to write hugely complicated routines to find projected intersection points, planes resulting from planar unions and or subtractions etc. AutoCAD's VBA programming interface was easy to work with and exposed all of the objects necessary to complete the IT package.

Delphi on the other hand proved a little more difficult to negotiate. Its programming environment requires a lot more discipline when writing code than VBA although this is rewarded by it picking up most mistakes at compile time rather than run-time. Delphi is designed for writing general applications and combining the internal objects and components with all the third party add-ons available the possibilities are endless. It was found to be the ideal tool to use for the non-drawing specific part of the IT package.

While an IT package was developed during the course of this work it was found that there would have to be an enormous amount of work to go into the user-interface end of the package. Should a useful product be developed for industry this work provides very useful building blocks for such a product. The user – interface with all the error handling required to tie all these blocks together for use by the manufacturing industry would take a considerable amount of time to develop.

10.2 Future Development

This completes the research to date. There is still one main aspect of the manufacturing process to be considered – the production process. In order to track a slab completely from inception to leaving the manufacturer's premises it is

necessary to monitor its production. This is envisaged by producing a graphical representation of the manufacturer's production beds. Slabs to be produced are dumped into production beds from where they are transferred to a database once produced. The production beds are dynamically linked to the database providing full tracking of the slab during the production process. The beds can show production linked to any date.

Since AutoCAD is already inherently a database all the information required to set up the production beds is already contained in the AutoCAD drawing. The production beds will thus query AutoCAD's database.

References

1: AutoCAD 2000 On-Line Documentation "AutoCAD ActiveX and VBA Reference"

2: AutoCAD 2000 On-Line Documentation "AutoCAD Command Reference"

3: Microsoft On-Line Documentation "Win32 Developer's References"

Delphi Win32api

4: AutoCAD 2000 On-Line Documentation "AutoCAD Visual Basic Reference"

5: Microsoft On-Line Documentation "Microsoft Data Access Objects Reference"

The location and the width of the supports can be changed. Consequently cantilevers can be set up. Selecting OK for a configuration recalculates and presents results for examination.
With the use of the toolbuttons described above it is possible to test the design of many slabs with the desire to satisfy the physical conditions of its use. Once an acceptable design combination has been achieved the user selects OK on the main form. Each design parameter that came from AutoCAD in the block is checked for a change and if one is applicable the appropriate change is written to the block. Finally the block's colour is changed to green to graphically indicate that it has been designed and the Design DLL is released to await another call!

Chapter 10 Conclusions

10.1 Achieved Developments

While developing this IT package the AutoCAD environment proved to be an extremely user-friendly versatile product to work with. Only a small subset of its vast drawing capabilities was used. The nature of the drawing objects provided by AutoCAD (such as the AcadLine and AcadRegion), with their methods and properties, avoided the need to write hugely complicated routines to find projected intersection points, planes resulting from planar unions and or subtractions etc. AutoCAD's VBA programming interface was easy to work with and exposed all of the objects necessary to complete the IT package.

Delphi on the other hand proved a little more difficult to negotiate. Its programming environment requires a lot more discipline when writing code than VBA although this is rewarded by it picking up most mistakes at compile time rather than run-time. Delphi is designed for writing general applications and combining the internal objects and components with all the third party add-ons available the possibilities are endless. It was found to be the ideal tool to use for the non-drawing specific part of the IT package.

While an IT package was developed during the course of this work it was found that there would have to be an enormous amount of work to go into the user-interface end of the package. Should a useful product be developed for industry this work provides very useful building blocks for such a product. The user – interface with all the error handling required to tie all these blocks together for use by the manufacturing industry would take a considerable amount of time to develop.

10.2 Future Development

This completes the research to date. There is still one main aspect of the manufacturing process to be considered – the production process. In order to track a slab completely from inception to leaving the manufacturer's premises it is

necessary to monitor its production. This is envisaged by producing a graphical representation of the manufacturer's production beds. Slabs to be produced are dumped into production beds from where they are transferred to a database once produced. The production beds are dynamically linked to the database providing full tracking of the slab during the production process. The beds can show production linked to any date.

Since AutoCAD is already inherently a database all the information required to set up the production beds is already contained in the AutoCAD drawing. The production beds will thus query AutoCAD's database.

References

1: AutoCAD 2000 On-Line Documentation “AutoCAD ActiveX and VBA Reference”

2: AutoCAD 2000 On-Line Documentation “AutoCAD Command Reference”

3: Microsoft On-Line Documentation “Win32 Developer’s References”

Delphi Win32api

4: AutoCAD 2000 On-Line Documentation “AutoCAD Visual Basic Reference”

5: Microsoft On-Line Documentation “Microsoft Data Access Objects Reference”