

UNIGRAPHICS

OPEN API
STUDENT GUIDE
February 2004
MT13110 – Unigraphics NX 2

EDS Inc.

Proprietary & Restricted Rights Notices

Copyright

Proprietary right of Unigraphics Solutions Inc., its subcontractors, or its suppliers are included in this software, in the data, documentation, or firmware related thereto, and in information disclosed therein. Neither this software, regardless of the form in which it exists, nor such data, information, or firmware may be used or disclosed to others for any purpose except as specifically authorized in writing by Unigraphics Solutions Inc. Recipient by accepting this document or utilizing this software agrees that neither this document nor the information disclosed herein nor any part thereof shall be reproduced or transferred to other documents or used or disclosed to others for manufacturing or any other purpose except as specifically authorized in writing by Unigraphics Solutions Inc.

©2003 Electronic Data Systems Corporation. All rights reserved.

Restricted Rights Legend

The commercial computer software and related documentation are provided with restricted rights. Use, duplication or disclosure by the U.S. Government is subject to the protections and restrictions as set forth in the Unigraphics Solutions Inc. commercial license for the software and/or documentation as prescribed in DOD FAR 227–7202–3(a), or for Civilian Agencies, in FAR 27.404(b)(2)(i), and any successor or similar regulation, as applicable. Unigraphics Solutions Inc., 10824 Hope Street, Cypress, CA 90630.

Warranties and Liabilities

All warranties and limitations thereof given by Unigraphics Solutions Inc. are set forth in the license agreement under which the software and/or documentation were provided. Nothing contained within or implied by the language of this document shall be considered to be a modification of such warranties.

The information and the software that are the subject of this document are subject to change without notice and should not be considered commitments by Unigraphics Solutions Inc.. Unigraphics Solutions Inc. assumes no responsibility for any errors that may be contained within this document.

The software discussed within this document is furnished under separate license agreement and is subject to use only in accordance with the licensing terms and conditions contained therein.

Trademarks

EDS, the EDS logo, UNIGRAPHICS SOLUTIONS®, UNIGRAPHICS®, GRIP®, PARASOLID®, UG®, UG/...®, UG SOLUTIONS®, iMAN® are trademarks or registered trademarks of Electronic Data Systems Corporation or its subsidiaries. All other logos or trademarks used herein are the property of their respective owners.

Open API Student Guide Publication History:

Version 15.0	February 1999
Version 16.0	February 2000
Version 17.0	May 2001
Version 18.0	March 2002
Unigraphics NX	February 2003
Unigraphics NX 2	February 2004

Table of Contents

Introduction	-1
How to Use this Book	-1
What is Open API?	-2
Two Forms of Open API	-2
Open API Applications	-3
Some useful applications using the Open API include: ...	-3
Course Objectives	-3
Some of the objectives for the Open API course include: .	-3
Open API Compiling and Linking	1-1
The UFMENU Utility	1-2
How to Invoke UFMENU	1-2
Edit	1-3
Compile	1-3
Compile Errors	1-5
Link	1-5
Link Errors	1-6
Run	1-6
Change Directory	1-7
List Directory	1-7
Non-menu Activities	1-7
Makefile Overview	1-8
Activity: Compile and Link example.c	1-10
Error Checking	1-11
Programming in Windows	1-12
A Second Method to compile and link in Windows	1-18
User Interaction	2-1
Overall Class Project	2-3
General Open API Program Format	2-5
License Check In/Check Out Routines	2-6
Calculator Entry Point, calculator.c	2-7
Code Discussion: calculator.c	2-7
Memory and Message Routines	2-9
Invocation Macro	2-9
Part Routines	2-11

Menu Routines	2–12
Code discussion: calc_part.c	2–12
Activity: Check Session for Open Part Files	2–15
User InterfaceStyler	2–17
Invoking the Styler	2–18
Styler Menus	2–19
User Interface Styler Resource Editor	2–20
Object Browser	2–21
Save Dialog	2–21
Open API routines for the User Interface Styler	2–22
Querying Attributes	2–22
Setting Attributes	2–23
Activity: User Interface Styler for Calculator Data	2–24
UI Routines for Object Selection	2–37
Assemblies	3–1
Top–Down Design	3–3
Assembly Terminology and Concepts	3–3
Sample Assembly	3–5
Coordinate Systems	3–10
Assembly Routines	3–11
Layer Routines	3–12
Object Routines	3–13
Code Discussion: calc_assem.c	3–14
Creating a Component	3–16
Activity: Create Assembly Parts for Calculator	3–17
Report Object Tags, Types, and Subtypes	3–18
Expressions	4–1
Expression Definition	4–2
Conditional Expressions	4–2
Expression Routines	4–4
Expressions for the calculator	4–5
Code Discussion: calc_expr.c	4–8
Activity: Create calc_expr	4–9
Modeling	5–1
Geometry Creation in Context	5–2
Model list routines	5–3
Code Discussion: calc_model.c	5–4
Activity: Create Routine calc_model	5–6
Calculator Top Half	5–7

Model Creation Routines	5–9
Model inquiry routines	5–10
Object Name Routines	5–11
Code Discussion: calc_model_top.c	5–12
Activity: Create the Calculator Top	5–17
Code Discussion: calc_model_top_holes.c	5–18
Activity: Add Holes to the Calculator Top	5–21
Calculator Bottom Half	5–22
Code Discussion: calc_model_bottom.c	5–22
Calculator Button	5–23
Code Discussion: calc_model_button.c	5–24
Activity: Create Bottom Half and Button Geometry	5–25
Code Discussion: calc_comp_array.c	5–26
Activity: Create Routine calc_comp_array	5–27
Dimensioning	6–1
Dimension and Drafting routines	6–2
calc_dimension.c	6–3
Code Discussion: calc_dimension_top.c	6–4
Code Discussion: calc_dimension_top_edges.c	6–6
calc_dimension_bottom.c	6–9
Activity: Create Top Dimensions	6–10
WCS control routines	6–12
View/Layout control routines	6–13
Code Discussion: calc_dimension_button.c	6–14
ug_set_wcs_to_view.c	6–15
calc_dimension_button_edges.c	6–15
Activity: Create Button Dimensions	6–16
Drawings	7–1
Member View and Drawing Name Conventions	7–1
Drawing routines	7–3
Importing Parts	7–4
Code Discussion: calc_drawing.c	7–4
Code Discussion: calc_drawing_views.c	7–6
Activity: Create a Drawing	7–8
External Open API	8–1
External Open API Programs	8–2
Attribute Routines	8–3
Adding Attributes to the Calculator Parts	8–3
calcx.c	8–4

Code Discussion: calcx.c	8-5
Activity: Attribute and External Open API	8-9
Plotting	A-1
Plotting in the Open API	A-1
Plot Routines	A-1
Code Discussion: calc_plot.c	A-1
Activity: Generating Drawing Plots	A-3
Glossary	GL-1
Index	IN-1

Introduction

This chapter introduces the Open API overall concepts, as well as the course layout. It also briefly details the two types of Open API programs, internal and external.

How to Use this Book

This Student Guide is intended as a classroom workbook. It is not a complete reference manual. For the full reference material, please refer to the API Reference Guide and the Open API Programmer's Guide.

For the API Reference Guide, choose Help → Documentation → Open API Reference Guide. (Notice that the Open C++ Programmer's Guide is in this same location.) You may also open this file and bookmark it in your browser: `${UGII_BASE_DIR}/ugdoc/ugopen_doc/main.html`.

The Open API Programmer's Guide is under the Open heading in the left column of NX Help.

Other course concepts are detailed in the Manuscript Users Guide and the User Interface Styler Help.

This Student Guide is arranged as a series of lessons. The routines created in each lesson are portions of an overall program. The program automates the design of a hand held calculator. Each lesson progresses the discussion of the necessary Unigraphics concepts and introduces the Open API tools used to construct the routines for the lesson activities. The Student Guide contains code templates and solutions for the activities.

What is Open API?

The Open API is a toolkit of C language callable routines for application programmers to write programs that work directly with Unigraphics. Unigraphics Solutions provides a library of C language functions, include files, and Fortran subroutines that interface with the Unigraphics object data base (Unigraphics Object Model).

The functions and header files can be used from C++ code, but this is currently beyond the scope of this course. Further information about using Open C/C++ programs is available in the Open API Reference and the Open C++ Programmers Guide, found under the directory ugdoc/ugopen++ where Unigraphics is installed, as file index.html.

Two Forms of Open API

Open API application programs can be developed in two forms: internal and external programs.

An *internal* Open API application program is a relocatable image that runs within the Unigraphics interactive process. Programs can be developed to access and create geometry, analyze geometry, create and edit features, create and edit expressions, and manage data. Routines to interact with users using standard Unigraphics Motif widgets are provided. Access to general Motif calls is supported within the Open API.

An *external* Open API application is a program that runs without displaying a graphics window. External Open API application programs tend to be more for data management than for geometry manipulation. External Open API application programs can construct and edit models, where no model display is necessary.

This training course will concentrate on using the internal Open C API, but will also include an external Open C API example.

Open API Applications

Some useful applications using the Open API include:

- Geometry/Feature creation and editing
- Expression creation/manipulation
- Geometry Analysis
- Part Standardization
- File management
- Data access
- Family of Parts
- Create and interact with User Interface Styler dialogs
- Create/maintain User Defined and Smart Objects

Course Objectives

Some of the objectives for the Open API course include:

- Create internal and external Open API programs using ufmenu or the make utility to compile and link
- Open, close, and save Unigraphics part files
- Develop User Interfaces using the User Interface Styler
- Create and edit features
- Demonstrate an understanding of solid/sheet body representations in the data model
- Create and manipulate expressions
- Work in context (working in assemblies)
- Create dimensions and plot drawings
- Learn to use Motif widgets in a Open API application program

(This Page Intentionally Left Blank)

Open API Compiling and Linking

Lesson 1



OBJECTIVES

Upon completion of this lesson, you will be able to:

- Invoke the ufmenu utility.
- Examine Edit, Compile, and Link options on the ufmenu.
- Generate a makefile from a template provided by Unigraphics.
- Compile, Link, and Run an internal User Function program using the ufmenu utility and the Unigraphics NX2 Open Wizard.

The UFMENU Utility

Ufmenu is a utility script/command file that provides you with the ability to edit, compile, and link your Open C and C++ API programs. Ufmenu uses the ufcomp utility to compile your programs and the uflink utility to link your programs.

The ufmenu window also allows you to:

- Run external Open API programs.
- Change directories.
- List the files in a directory.
- Spawn a new process.
- Quit (exit) the programming environment.

NOTE Ufmenu is supported only on Unix systems. If you use Windows, you need to use a Windows based editing and compiling system. Please refer to the topic Programming in Windows , at the end of this lesson.

How to Invoke UFMENU

Ufmenu is invoked when you select the OPEN API (User Function) option from the Uniproducts Menu. After you invoke ufmenu, the User Function Development Environment menu option displays.

```

+-----+
| USER FUNCTION DEVELOPMENT ENVIRONMENT |
+-----+

1) Edit                5) change Directory
2) Compile             6) liSt directory
3) Link                7) Non-menu activities
4) Run (external user function)  q) Quit

```

Enter option (1-7, q) [q]:

The material in this section is a basic discussion of compiling and linking. The Open API documentation gives a complete discussion of compiling and linking for each platform. Internal Open API programs must, on UNIX platforms, be generated using the position independent code (PIC) compiler switch. This is *not* included as a part of the standard compiler. A compiler upgrade may be necessary to be able to generate PIC output on your code development system.

The following paragraphs describe the options found on the UNIX version of `ufmenu`. You can select options by entering the number of the option or by entering the capitalized letter in each option name as it appears in the main `ufmenu`. For example, to list the contents of your current directory, you can either enter option number **6** or the letter **S** (**liSt directory**).



Edit

The `edit` option allows you to edit a file with the currently specified operating system editor (for example, `vi`). The default editor is specified by the `UGII_EDITOR` variable for UNIX. If the file does not reside in your current directory, enter the full directory path for the file specification. You can include the file extension (`.c`) or use a wildcard.

```
Enter option (1–7,q) [q]: 1
Enter file(s) to edit (vi) [block1.c block2.c block3.c bounded_plane.c
testopen.c]: bounded_plane.c
```

Compile

The `compile` option invokes the C or C++ compiler which converts the statements of your C or C++ source file into an object file (`.o`). You can specify a file template, such as `*.c`. This compiles all the files in your current directory with the appropriate file extension. An ANSI compiler is strongly recommended. An ANSI C compiler is required for C language Open API application programs if the UG Solutions provided header files (which include prototypes) are used.

The compiler will generate an object file for each program successfully compiled. On UNIX platforms, the `.c` will be replaced by `.o`.

You must include the file extension for your source programs. You can compile more than one file by delimiting your file names with a space. The `compile` option automatically determines the appropriate compile options for your platform. A list of all the files with a `.c` extension appears within square brackets. For example:

```
Enter option (1–7,q) [q]: 2
Enter file(s) (separated by “”) to compile [block1.c block2.c block3.c
bounded_plane.c testopen.c]: bounded_plane.c
```

```

Compiling... bounded_plane.c

Default C compile options: -c -KPIC -Xc -I. -I<path>
Change compile options (y/n) [n]: n

bounded_plane.c compiled successfully.

Hit <RETURN> to continue.

```

The default compile option switches can be the same for both internal and external Open API application programs. The switch for Position Independent Code can be used for external User Function. However, if you wish to change any of the default options, enter “y” to the prompt: “Change compile options (y/n) [n]:”. Additional “-I” switches can be added to allow the compiler to find Motif, X11, or other header files.

When you enter “y” to the Change Options prompt, the ufmenu script prompts you for the mode (internal or external) and which compile options to remove. It then prompts you for options to add to the compile command line. In the following example, we show how to change compile options and add the `-g` option to compile for debugging.

```

Enter option (1-7,q) [q]: 2
Enter file(s) (separated by “ ”) to compile [block1.c block2.c block3.c
bounded_plane.c testopen.c]: testopen.c block1.c block2.c block3.c

Compiling... testopen.c block1.c block2.c block3.c

Default C compile options: -c +Z -Aa -I. -I/usr/ugs160/ugopen
Change compile options (y/n) [n]: y

Compile internal/external user function (i/e) [i]: e
Remove +Z (y/n) [n]: y
Remove -Aa (y/n) [n]:
Remove -I. (y/n) [n]:
Remove -I/usr/ugs160/ugopen (y/n) [n]:
Add new options: -g -I/user1/include
New compile options: -c -Aa -I. -I/usr/ugs160/ugopen -g
-I/user1/include

testopen.c compiled successfully.
block1.c compiled successfully.
block2.c compiled successfully.
block3.c compiled successfully.

Hit <RETURN> to continue.

```

Compile Errors

If your compile should fail, ufmenu creates an error log file in the current directory. The name of log file is of the form “username<pid>.complog”. On UNIX systems each compile is a different process so one complog file is created per compilation. The script displays a message similar to the following:

```
block1.c did not compile. Refer to username6370.complog for error
message.
```

The indicated .complog file can be examined to determine the problems found with the source code files, so that correction can be made.

Link

The link option allows the linking the object file of a primary C or C++ program with the object files of any subprograms which can be referenced. The link option calls the uflink utility (refer to the Open API Reference documentation for complete details). The main object file must reside in your current directory. You can use a directory specification for your subroutines. Ufmenu automatically invokes the uflink script. The following example links an external User Function image in debug mode.

```
Enter option (1-7,q) [q]: 3
Link internal/external user function (i/e) [i]: e
Link a C++ image (y/n) [n]: n

Default uflink options: -m
Change uflink options (y/n) [n]: y
Add new options: -d
New uflink options: -m -d

Enter program to link => testopen
Enter any subroutines => block1.o block2.o block3.o
Enter any libraries =>

uflink:WARNING - UGII_USERFCN variable not set.
Using libraries in <path> as a default.

Linking with: block1.o block2.o block3.o.
/bin/cc options: -Wl,-q,-E,-B,immediate,+s,-L,<path> -g
Linking... testopen for external execution.
```

```
uflink:link SUCCESSFUL – Wed Jan 26 10:33:07 PDT 2000
```

```
Hit <RETURN> to continue.
```

The link step will produce an executable file if there are no errors. The file name on UNIX platforms is the name entered at the prompt “Enter program to link =>”. A file with the program name and a “.map” extension will also be created. This file is used when the `-m` option (map) is specified as a `uflink` option (see the Open API Reference manual for more details on the `-m` option).

Link Errors

The link script will display a message indicating the success or failure of the link step. A file with the program name and `.errors` will be created in the current directory. It will contain the error text generated by the linker. You will not receive errors regarding unresolved references (you may receive warnings on some platforms). Internal and External user functions resolve all calls upon invocation.

Run

The run option allows you to execute an external Open API application program. The following example shows how an argument is passed to the program (similar to invoking the program from a shell prompt with `argument(s)`).

```
Enter option (1–7,q) [q]: 4
Enter external user function to run [testopen]:
Run debug mode (y/n) [n]: n
Enter arguments to pass to testopen []: newbox
Hit <RETURN> to continue.
```

The next example shows the prompts when you run in debug mode.

```
Enter option (1–7,q) [q]: 4
Enter external user function to run [testopen]:
Run debug mode (y/n) [n]: y
Copyright Hewlett–Packard Co. 1985,1987–1992. All Rights
Reserved.
<<<< XDB Version A.09.01 HP–UX >>>>
No core file
Procedures: 4
Files: 4
testopen.c: main: 15: int units = 2;
>
```


From here, you can now enter debug commands. The debugger and debugger prompt are platform specific.



Change Directory

The Change Directory option allows you to change the current directory by entering a new directory pathname.

```
Enter option (1–7,q) [q]: 5
Current directory => /class/userfunc/test
Enter new directory [.]: /class/userfunc
New directory => /class/userfunc
```

List Directory

The List Directory option allows you to list the contents of the current directory. You can specify a file template. When you choose this option, `ufmenu` displays your current directory and prompts you for a template of the files to list. The default template is to list all files. You can enter any valid wildcard specification.

```
Current directory => /class/userfunc
Enter file(s) to list [*]:
```

Non–menu Activities

This option causes `ufmenu` to spawn an operating system shell as a child process. The script prompts you for the shell type as follows:

```
Enter Shell Type (sh/csh/ksh) [ksh]:
```

The default value is the Korn shell (`ksh`). You can spawn a Bourne shell (`sh`) or a C–shell (`csh`) by entering the appropriate value. For example, entering `csh` will spawn a C–shell.

Makefile Overview

The Unix make utility can be invoked from the Unix shell command line in any window (with appropriate Unigraphics environment variables set). The provided makefile template can be used with the make utility to compile and link internal and external Open API programs.

The make utility checks the last modified date on specified source and dependent files (such as header files). The make utility will regenerate an executable program if source or dependent files have been modified.

On Unix systems (HP, SUN, Digital Unix, SGI, and AIX), a template makefile (ufun_make_template.ksh) is provided in the $\{\text{UGII_BASE_DIR}\}/\text{ugopen}$ directory. The template can be copied and modified to compile and link your Open API programs. The makefile, like the ufmnu script, uses the ufcmp and uflink scripts to compile and link Open API programs.

The template makefile can be customized to compile and link the following possibilities: both internal and external, internal only, or external only Open API programs. Instructions on how to use the template makefile are fully explained in the comments section of the file.

When you customize the template file, you should copy or move the file so that it is named either "Makefile" or "makefile". These are the standard names the make command expects. Alternatively, you could use the "-f" switch with the make utility to specify another makefile name that is not of the default name format (e.g. "make -f my_program_makefile").

The template makefile was designed to be used with the make command supplied by the platform's vendor (HP, SUN, etc.). For further information on make and file dependencies use, consult your operating system's provided utilities documentation ("man make").

To create an internal Open API program, you would follow the directions from the makefile. The first step is to set the variable that defines the executable name of the program. This should be the lead routine (without the .o suffix).

The next step is to list any subroutine object files. This list must be continuous on ONE line only. The list can be continued to subsequent lines by using the \ character at the end of line. You must be careful that the \ is the last character on the line (no spaces after the \). All routine names should have the .o extension.

The C compile flags can (optionally) be specified next. Note that to add flags, you **must** specify the `-ac` switch. The flags can be specified in quoted string (e.g. `-ac "-g -I/usr/include/Motif1.2 -I/usr/include/X11R5"` can be used to compile with the debug switch and to add the two directories to resolve include files).



The Fortran flags are also available. Fortran callable C wrappers can be used to call Open API functions if you must use Fortran.

The link flags are available for change. These are equivalent to the data obtained using the `ufmenu link` option.

Any of your own libraries or those of third party applications that are needed by the Open API program are listed here.

The make utility has rules to determine which files are to be compiled and linked. The executable must be recreated when object files change. Object files need to be created when source files change. These rules are implicit in make. However, object files may also need to be recreated when header files are changed. Explicit rules allow these situations to be defined.

Activity: Compile and Link example.c

1

Step 1 Create a sub–directory using your initials or name. You will perform all activities in your sub–directory.

Step 2 The following code is contained in a file called *example.c*. Please copy this into your sub–directory. Use *ufmenu* to compile and link the program.

Step 3 Run Unigraphics and execute the program. Try varying number of parts open in your session, and the units of open parts.

Error Checking

We strongly recommend that you include this macro definition in your header file and incorporate the error checking macro in every Unigraphics function call *that returns a non-zero integer to indicate an error*:

```
#define UF_CALL(X) (report_error( __FILE__, __LINE__, #X, (X)))

static int report_error( char *file, int line, char *call, int irc)
{
if (irc)
{
char err[133],
msg[133];
sprintf(msg, "*** ERROR code %d at line %d in %s:\n+++ ",
irc, line, file);
UF_get_fail_message(irc, err);
/* NOTE: UF_print_syslog is new in V18 */
UF_print_syslog(msg, FALSE);
UF_print_syslog(err, FALSE);
UF_print_syslog("\n", FALSE);
UF_print_syslog(call, FALSE);
UF_print_syslog(";\n", FALSE);
if (!UF_UI_open_listing_window())
{
UF_UI_write_listing_window(msg);
UF_UI_write_listing_window(err);
UF_UI_write_listing_window("\n");
UF_UI_write_listing_window(call);
UF_UI_write_listing_window(";\n");
}
}
return(irc);
}
```

Example

If the function returns an integer as an error flag, instead of this call:

```
UF_PART_save_all( &errcount, &errtags, &errcodes);
```

Use this call:

```
UF_CALL( UF_PART_save_all( &errcount, &errtags, &errcodes) );
```

If a run time error occurs you will see a listing window with text pointing to the line of code where the error occurred. Notice that the function called by the macro checks all return values. Any non zero error code is used to exit via *return (irc)*.



Programming in Windows

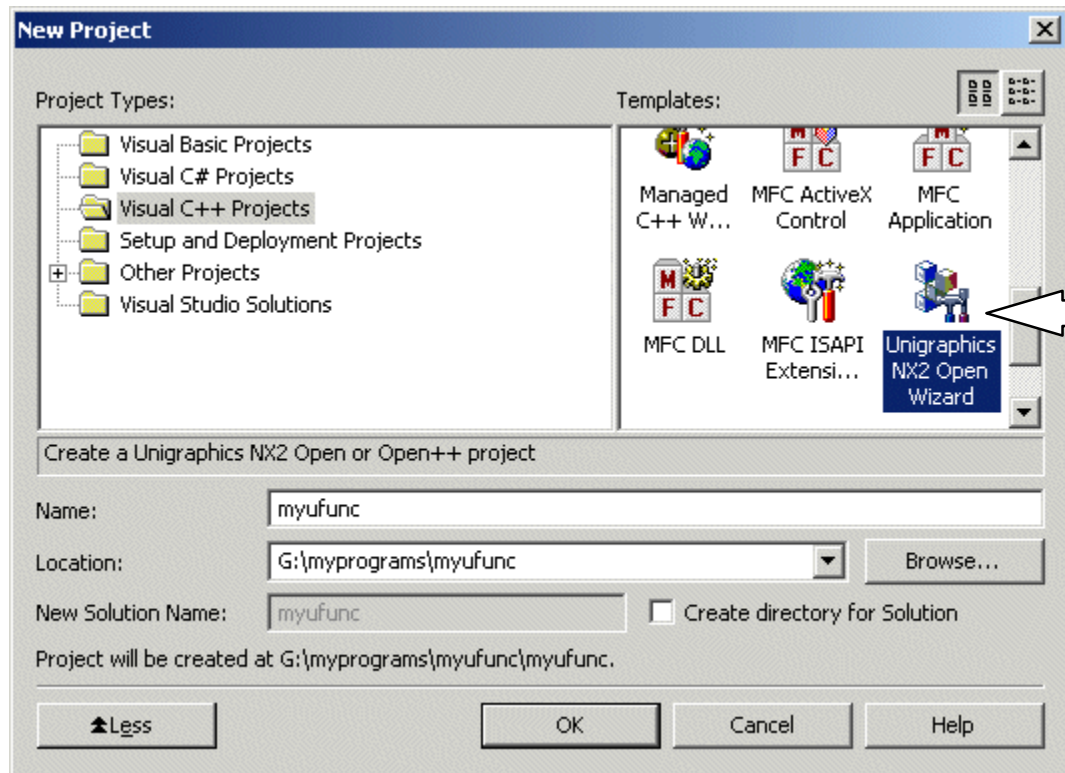
Microsoft Visual C++ .NET V7.0 was used to verify the programming examples in this course.

To use Visual C with the best results, perform the following steps. These steps assume that that Visual C++ is installed at
C:\Program Files\Microsoft Visual Studio .NET\Vc7 :

1. Copy all files from
 UGII_BASE_DIR\ugopen\vc7_files\vcprojects
 to
 C:\Program Files\Microsoft Visual Studio .NET\Vc7\vcprojects
2. Copy the entire folder
 UGII_BASE_DIR\UGOPEN\vc7_files\VCWizards\Unigraphics_NX2_Open
 to
 C:\Program Files\Microsoft Visual Studio .NET\Vc7\VCWizards
3. Set (or Edit) the Environment variable:
 MSVCDir
 to
 C:\Program Files\Microsoft Visual Studio .NET\Vc7
4. Start a Unigraphics NX 2.0 Command Prompt window
5. Start Microsoft Visual C++ from this window using the command "devenv"

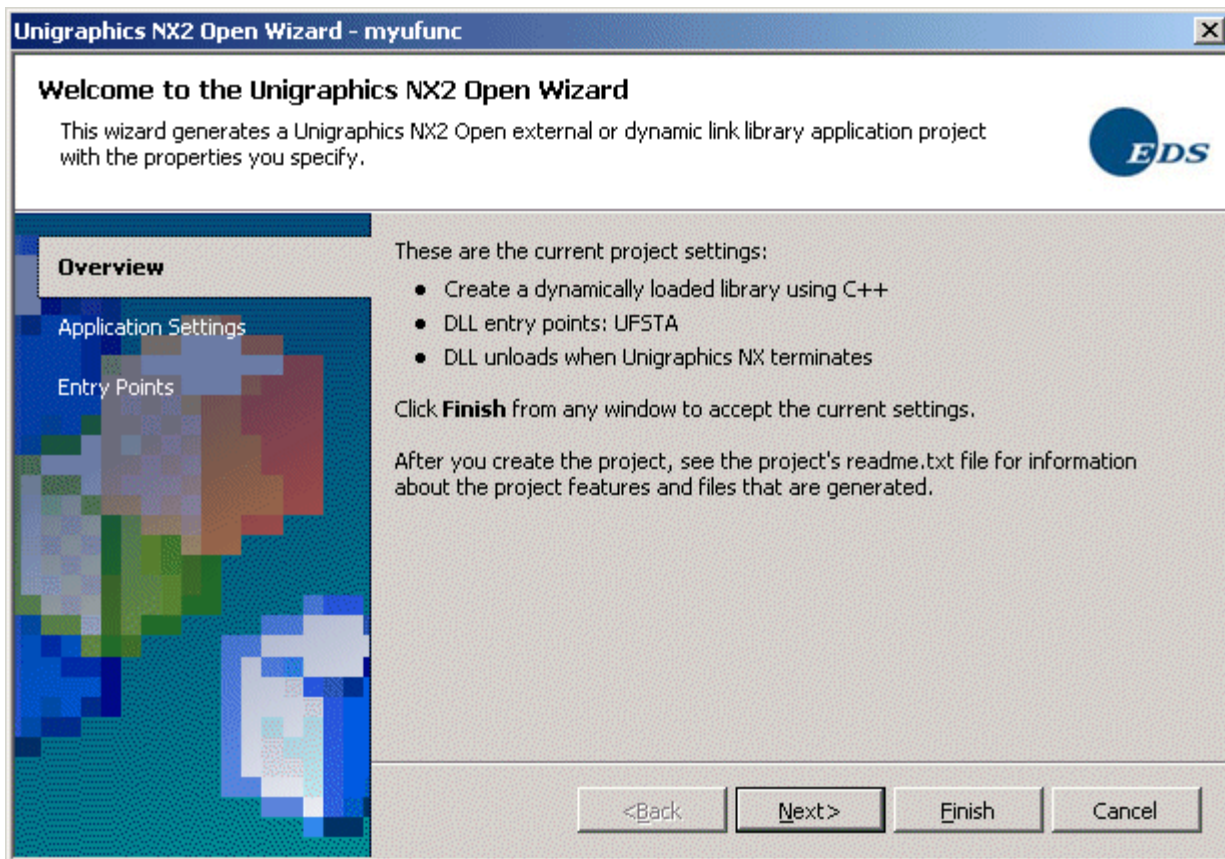
When you create a new project, a project template for the Unigraphics NX 2 OpenWizard will be available.

Fill in a project **Name**, **Location**, other options if applicable, and choose **OK**.

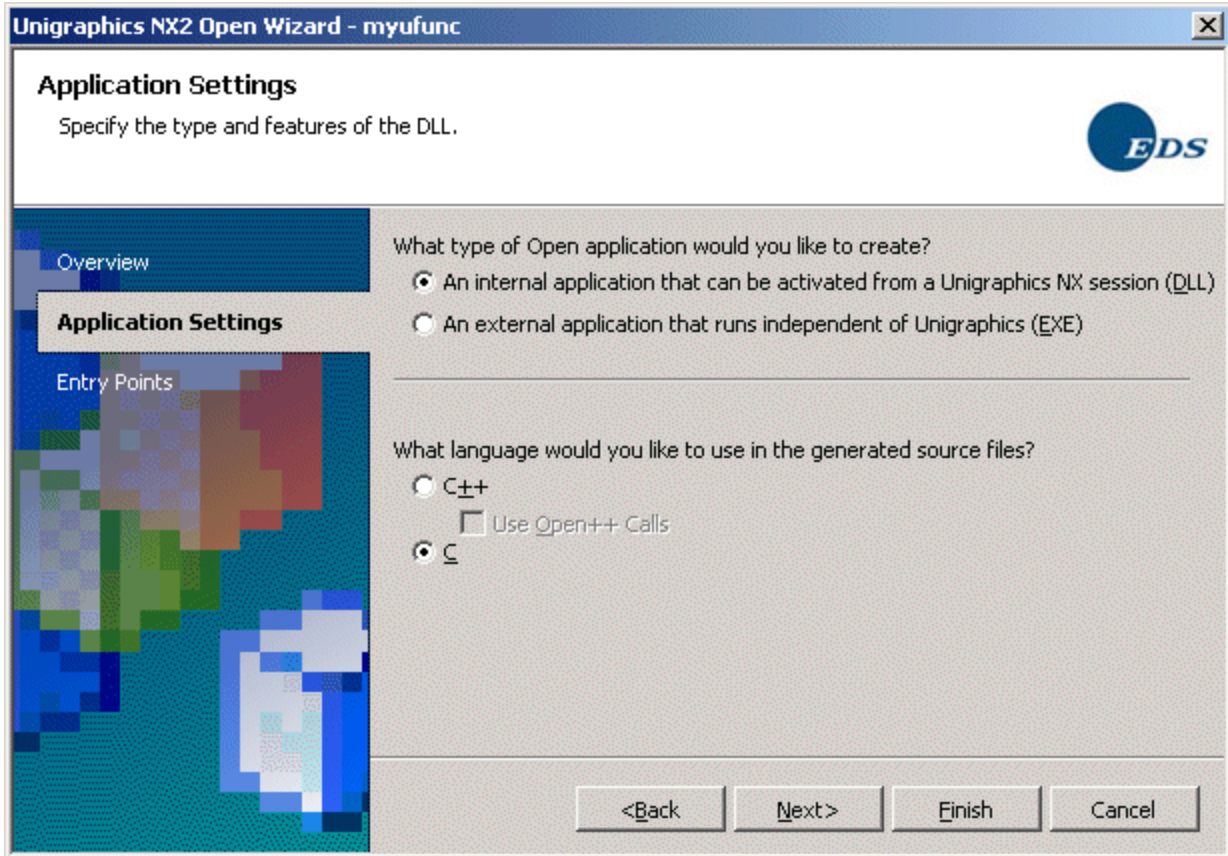


The **Unigraphics NX2 Open Wizard** overview window will appear and list the default project settings. Choose **Next** to change the project settings.

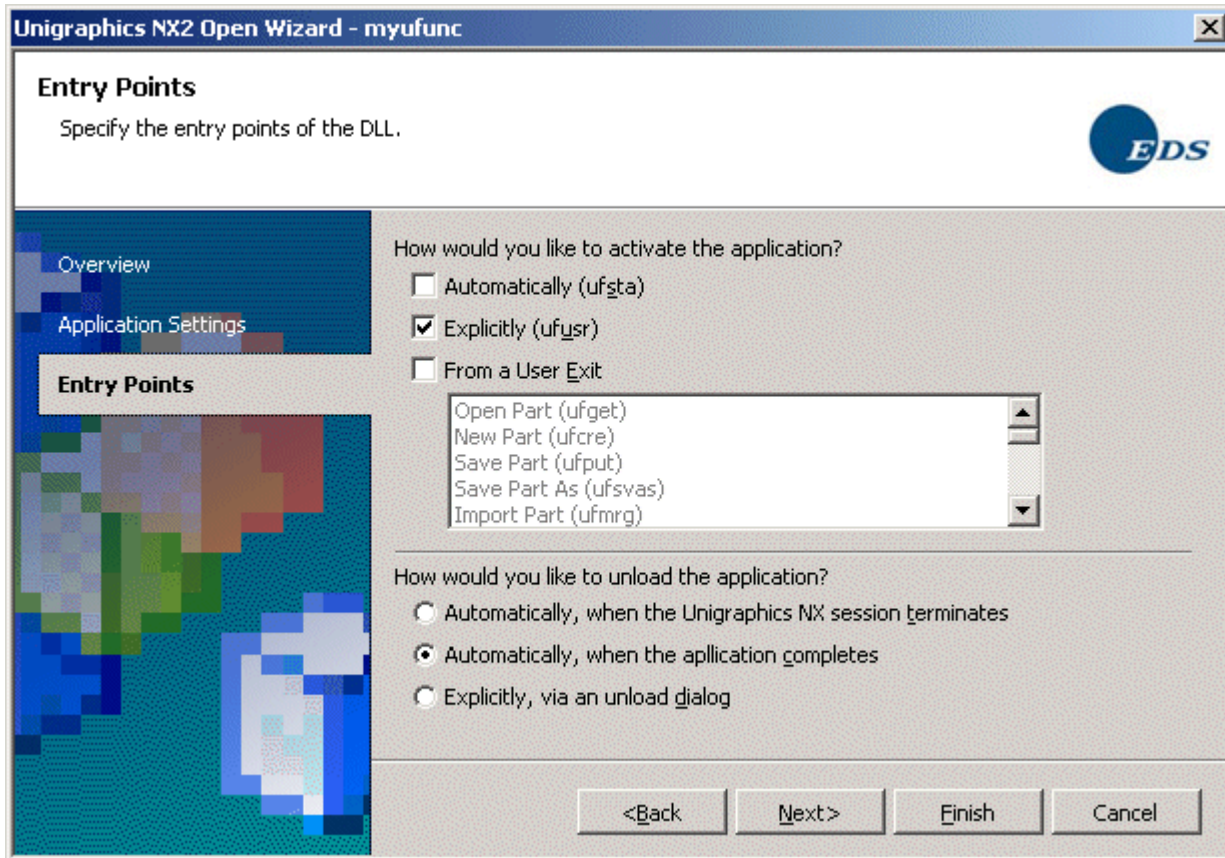
1



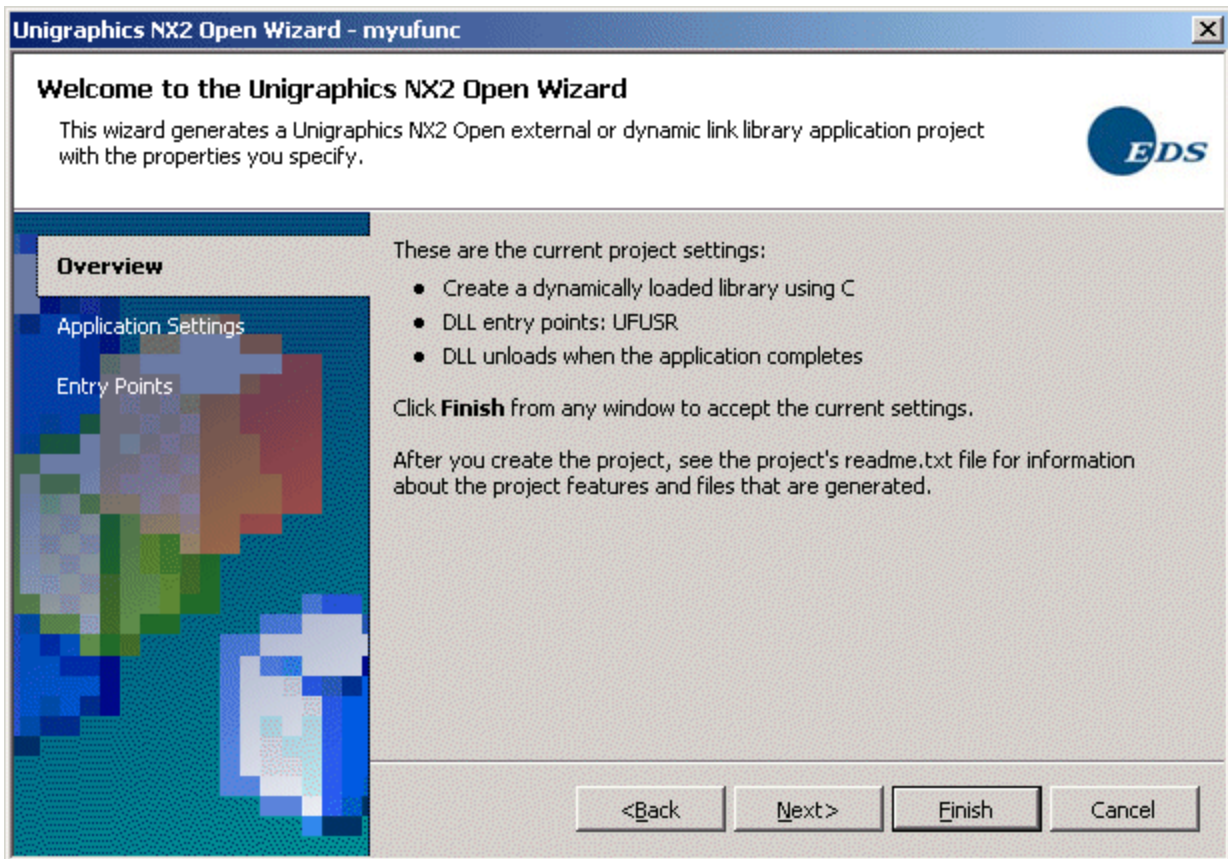
In the Applications Settings window, choose the type of application (internal or external) and the source code type (C++ or C). Choose **Next** to specify Entry Points.



In the Entry Points window, specify one or more activation options, a User Exit if applicable, and an unloading option. When you have selected the desired options, choose **Next**.



The Overview window appears again, listing the options that were specified for the project.

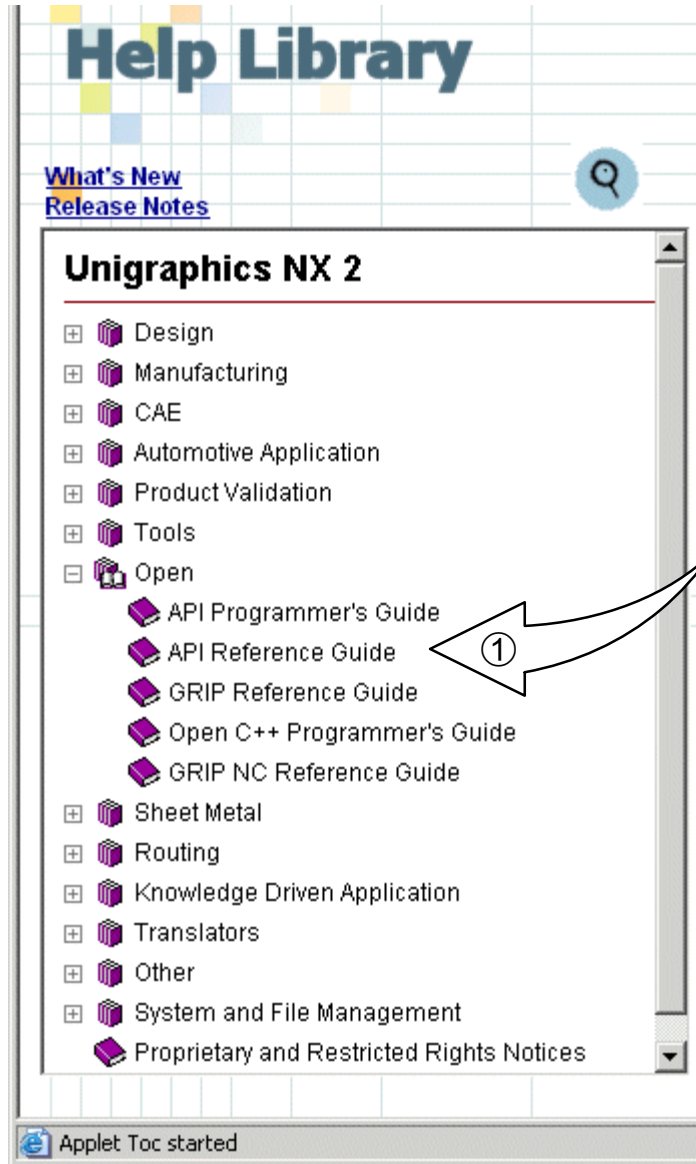


When you choose **Finish**, the Unigraphics NX2 Open Wizard will create a project containing a C source file. The source file will contain comment lines indicating where to place your code.

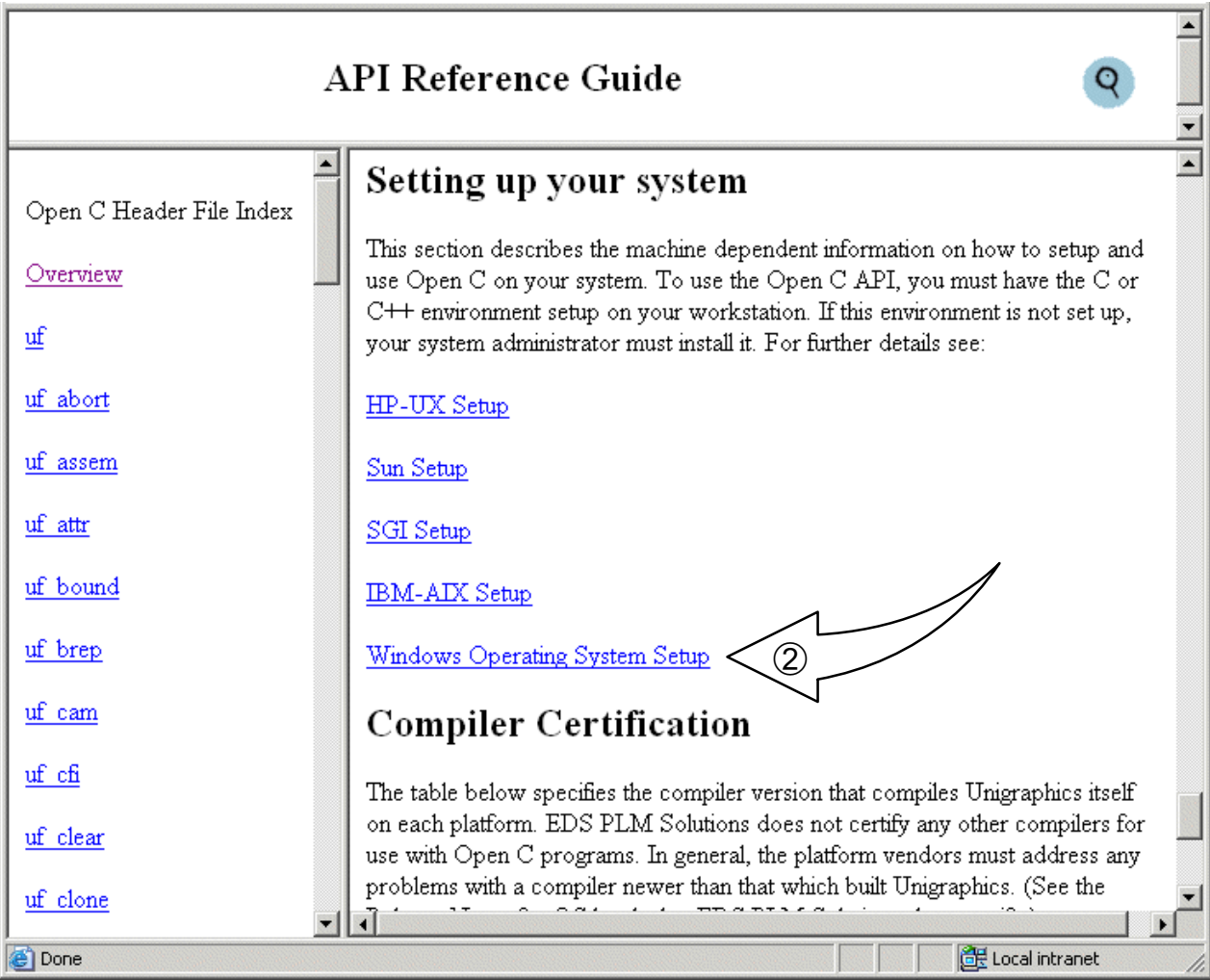
A Second Method to compile and link in Windows

If you have proven code, you can compile and link from the DOS command line. To find information about this use the following sequence.

Start the Help application and choose the Open API Reference Guide ①:



Scroll the Overview section of the API Reference Guide to **Windows Operating System Setup** ②:



Finally, in the Windows Operating System Setup section, scroll to the heading **Developing From the Command Line**.

SUMMARY

You have examined both methods of compiling and linking Open API programs, using either the ufmenu interface or the make utility.

In this lesson, you :

- Invoked the ufmenu window.
- Examined the options available on the ufmenu for Edit, Compile, and Link.
- Learned about the makefile template provided by Unigraphics.
- Compiled, Linked and Ran an internal User Function program.

User Interaction

Lesson 2

An important aspect of Open API program development is the interaction with the Unigraphics user. Unigraphics provides both callable functions (Open API) and an interactive Graphical User Interface (GUI) builder (User Interface Styler) to interface with the program user. The available functions consist of both Open API legacy functions and User Interface Styler API functions.



OBJECTIVES

Upon completion of this lesson, you will be able to:

- Check out and check in Open API licenses.
- Control how Unigraphics loads and unloads programs in memory.
- Identify routines to open, close, and save Unigraphics part files.
- Invoke and use the User Interface Styler Dialog to build custom dialogs.
- Work with the files created by the User Interface Styler Dialog.
- Obtain and set data from a User Interface Styler dialog.

The code templates and the necessary Open API routines to create the shell components of a hand-held calculator will be presented. Please use your student sub-directory to edit and create programs/parts. When you complete the project, you will have created the following functions:

- calculator.c
- calc_assem.c
- calc_comp_array.c
- calc_dimension.c
- calc_dimension_bottom.c
- calc_dimension_button.c
- calc_dimension_button_edges.c
- calc_dimension_top.c
- calc_dimension_top_edges.c
- calc_drawing.c
- calc_drawing_views.c
- calc_edit.c *
- calc_expr.c
- calc_model.c
- calc_model_bottom.c
- calc_model_button.c
- calc_model_top.c
- calc_model_top_holes.c
- calc_part.c
- calc_plot.c *
- xxx_calc_setup.c
- calc_set_style_dialog.c
- ug_set_wcs_to_view.c

* optional exercises

Overall Class Project

The top–level function for the internal Open API application executable will be **calculator.c**. You will copy the template version of **calculator.c** from the default directory to your sub–directory and edit your copy. This same scheme will apply to all the other functions created for this program. Please note that the code is written specifically for the UNIX workstation environment, although most of it should transfer readily to an Windows environment.

Additionally, there may be further activities in each lesson that create subsequent Open API programs to accomplish various tasks. Although unrelated to the Calculator project, these tasks are meant to give a broader understanding of the Open API routines available.



There are three main components in the calculator assembly model: the top half, bottom half, and button. The holes in the top half are created with rectangular pocket features. The button hole is duplicated using a feature instance array. Multiple copies of the button component are added as a component array.

At the end of each activity, compile, link and run the latest version of the program to make sure that there are no errors. Be sure to check the appropriate compile log files for warnings and errors.

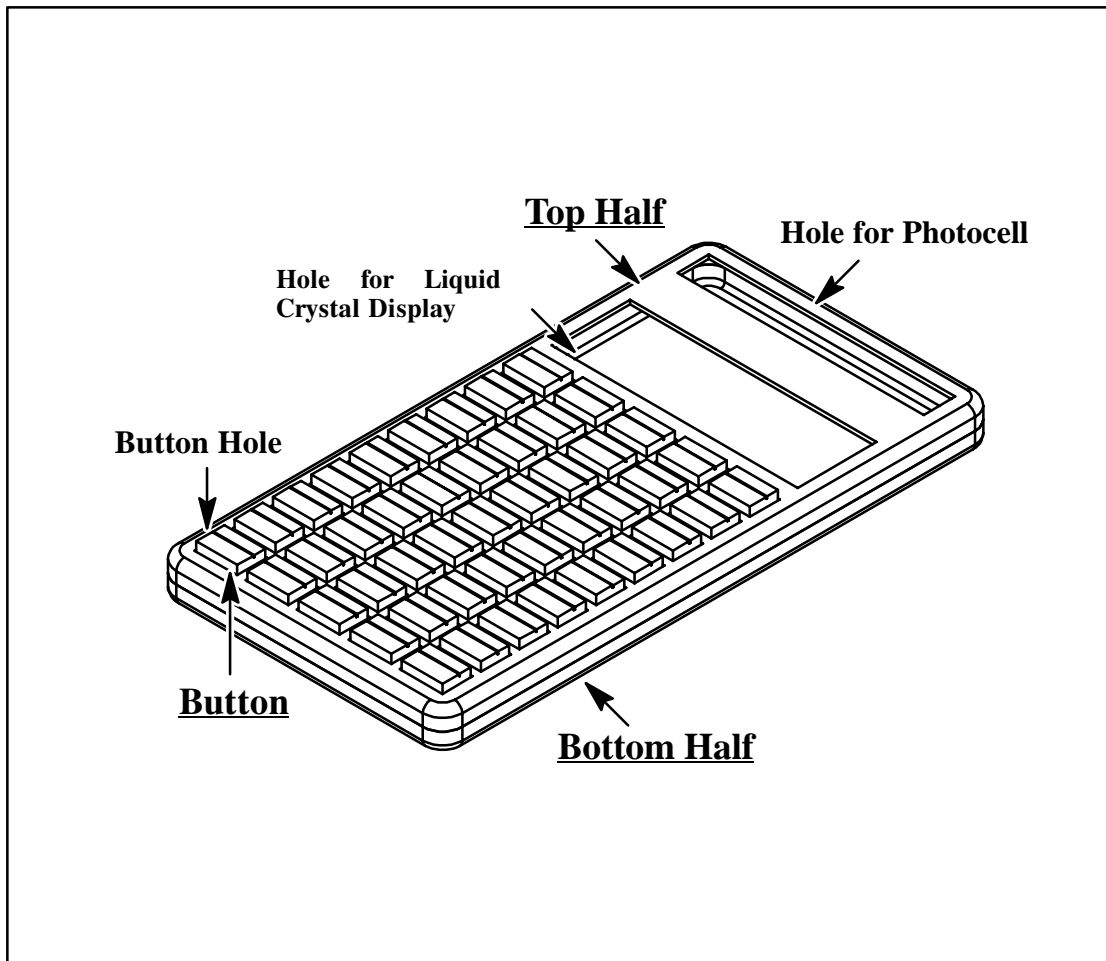


Figure 2–1 Model of hand–held calculator

The first two activities will address the initial program setup and user interaction. Internal Open API programs use an entry point named *ufusr*. The routine name is prototyped in *uf.h*. All executable C language internal Open API programs should start with *ufusr* and include the header file *uf.h*. The three *ufusr* parameters are a character pointer, integer pointer and integer argument. Although not currently used in our courseware activities, these three input parameters must still be declared.

General Open API Program Format

```
# include <uf.h>
void ufusr (char *param, int *retcode, int param_len)
{int irc;
irc = UF_initialize();
check return code; Body of program;
irc = UF_terminate();}
int ufusr_ask_unload(void)
{return (UF_UNLOAD_IMMEDIATELY);}
```

Once an internal Open API program is loaded into memory, it remains resident until Unigraphics is exited. This default behavior can be changed to force the program to unload after usage or unload when selected by the user. The function called *ufusr_ask_unload* can be defined for these actions by the programmer. Unigraphics will call this routine if it has been defined. You should **NOT** call the routine from your program.

When you are developing code, you will normally use the “unload immediately” option. This allows you to make changes to your source and reload the updated shared image without restarting Unigraphics. If the unload option is not used, the original program executable image remains in use until you exit Unigraphics.

The initialize and terminate routines check out and check in, respectively, a Open API license. These must be present in any Open API program or Open User Exit routine.



License Check In/Check Out Routines

A Open API program must check out a license before any calls to Open API routines are made.

- **UF_initialize**
- **UF_terminate**
- **int UF_terminate(void)**
- **ufusr_ask_unload**
- **ufusr_cleanup**

NOTE You must use `ufusr_cleanup` in conjunction with `ufusr_ask_unload`. If you code `ufusr_cleanup` without defining `ufusr_ask_unload`, then the `ufusr_cleanup` entry point is ignored.

Throughout the class, functions of interest will be listed in this manual. Look up the corresponding reference material using a web browser on your workstation.

The API Reference Guide

Open the file `.../UGDOC/html_files/ugopen_doc/main.html` and choose the header file **uf** from the list at the left of the window. In the window that opens, choose **Functions** and refer to the documentation on each of the above items.

Calculator Entry Point, *calculator.c*

Our Open API entry point will be declared in the file *calculator.c*. The program has comments regarding the steps that will be taken to create the calculator. Code will be added to this program in the course activities to perform the commented tasks.

Code Discussion: *calculator.c*

The first line in the program includes the main Open API header file.

```
#include <uf.h>
```

```
#include <uf_defs.h>
```

uf.h contains the prototypes for the `UF_initialize` and `UF_terminate` functions. Other general purpose routines are also prototyped in this header file. The documentation of each routine in the Open API Reference has the appropriate header file with the routine's prototype listed on the starting page of the manual section.

The next include file, **uf_defs.h**, contains typedefs for general Open API programming. User Function (the legacy name of Open API) was originally written for Fortran. For cases where programs are migrating to V16 from earlier version of Unigraphics, routines for replacing the Fortran calls are documented in various text files (search on “legacy”) in the `ugopen` directory.

The Open API Reference contains a complete list of all header files. Each chapter that introduces a group of functions indicates which include files are needed. The header files should be located when Unigraphics is installed in the `Open` sub-directory (`$UGII_BASE_DIR/ugopen`).

```
#include "calcproto.h"
```

```
void ufusr(char *message, int *eflag, int mlen)
```

The file “`calcproto.h`” is our function prototype file, which we will build as the program is developed. The three arguments in the `ufusr` declaration, `message`, `eflag`, and `mlen`, are not used by the Open API. These arguments are used by programs written as User Exits.



Checking for an active part is the first activity to be performed in the program (after calling `UF_initialize` successfully).

```
/* check for an active part */
   flag = calc_part();
   if(flag != 0) return ( );
```

Most programs you will write will want to know if a part is loaded. Some programs may load a part if necessary. The function *calc_part* is outlined in subsequent pages.

Memory and Message Routines

Some Open API routines will allocate memory for data that persists once the function has returned. The Open API does not directly use the regular operating system provided malloc routines for this memory allocation. Unigraphics has its own memory manager which efficiently reserves and frees memory. Routines that allocate memory will have an indication in the I/O column of the documentation of **OF**. This indicates that the variable is a persistent output variable and must be freed. Unless otherwise specified, the routines `UF_free` and `UF_free_string_array` should be used to free the Unigraphics memory manager allocated memory before the program terminates.

Memory allocated by any Open API routine will stay allocated until the appropriate memory freeing routine is called or until the Unigraphics session is terminated. You must be careful to free any allocated memory, especially in programs or routines that may be run often during a Unigraphics session, otherwise there could be memory leakage problems, resulting in an “Out of Memory in Storage Manager,” or similar error.

The routines that allocate memory will be evident by the fact that a pointer to a pointer will be required for the Open API function argument. The I/O column in the calling argument table will have the mnemonic **OF** for Output/Free memory. A general convention in the Open API Reference manual is for typedefs of pointers to include an **_p** in their name. The variable `tag_t` is a tag of an object or part. A `tag_p_t` is a pointer to the tag of an object or part (`tag_t *` is equivalent to `tag_p_t`).

- `UF_free`
- `UF_free_string_array`
- `UF_get_fail_message`

Invocation Macro

Many Open API functions return integer values that can be recognized by `UF_get_fail_message`. For those routines, the process of checking the error code and printing the message can be done using a macro (C language preprocessor macro; not to be confused with a Unigraphics Macro). The macro is designed for **only** routines that return integers recognizable by the error message routine.

```
#define UF_CALL(X) (report( #X, __FILE__, __LINE__, (X)))
```



The `UF_CALL` macro invokes a subroutine (below) with the last argument (X) being the actual Open API call. The C preprocessor directives for the file name and line number have two underscores before and two after the words (FILE and LINE).

```
static int report( char *call, char *file, int line, int irc)
{if (irc) {char   messg[133];
  printf("%s\n%s, line %d: ", call, file, line);
  (UF_get_fail_message(irc, messg)) ?
  printf("Returned %d\n", irc) :
  printf("Error %d: %s\n", irc, messg);}
return(irc);} /* report */
```

The report routine is static and will print out a message if a non-zero return code is received. It also returns the code to the calling routine. The `#define` and report routine can be added to `calcproto.h` and used in the class exercises.

Part Routines

Unigraphics Open API provides many routines to access information about part files. You can determine the number of files open, the name of all open part files, the part file units, and the display part. You have the ability to open, close, export and import part files. The routines are prototyped in the header file `uf_part.h`. The header file also contains the definition of a structure used when opening part files. The routines we will be using are discussed on the following pages.

- **UF_PART_open**
- **UF_PART_close_all**
- **UF_PART_save_all**
- **UF_PART_ask_part_name**
- **UF_PART_ask_part_tag**
- **UF_PART_ask_units**
- **UF_PART_ask_display_part**
- **UF_PART_set_display_part**
- **UF_PART_is_modified**



Menu Routines

All user interaction can be performed through the Open API library of functions. The interaction functions currently have names that begin with uc16xx. User Interface Styler and Motif programs can also be incorporated with Open programs to perform user interface tasks.

The User Interface Styler routines require a Styler license. The uc16xx functions have limitations that are not present in the Styler functions. Some of the older functions will be covered but our main focus will be on the User Interface Styler.

- **uc1601** **Display a Simple Message on the Cue line**
- **UF_UI_set_status**
- **UF_UI_set_prompt**
- **uc1603** **Display Selection Menu**
- **uc1605** **Multiple Selection Menu**

NOTE Please avoid using format escape sequences in the cp1 character string in uc1601. For example, the newline escape sequence (\n) can cause an undesirable shift in the displayed text.

Code discussion: *calc_part.c*

```
tag_t parent;      /* tag of parent part */
```

tag_t is a type definition used by Unigraphics. Currently, a *tag_t* is an unsigned integer. Use “tag_t” whenever you are creating a variable for an object; whether the object is a part or a piece of model geometry. If the typedef should ever change, your code will automatically change with it. If you use “unsigned int” rather than tag_t, you would have to change your code when (if) the typedef for tag_t is changed.

```
numparts = UF_PART_ask_num_parts();
```

Count the parts that are loaded.

```
resp = uc1603(estr,deft,option,2);
```

uc1603 presents a menu to the user.

```
UF_PART_save_all(&errcount,&errtags,&errcodes);
```

Save all loaded parts. Note that the arguments to this function are declared as:

- int errcount
- tag_t *errtags
- int *errcodes

errtags and **errcodes** are allocated by *UF_PART_save_all* and becomes an array of part tags of the parts that could not be saved and an array of error codes. If the return from *UF_PART_save_all* is not zero, these arrays should be freed with *UF_free* once the appropriate actions to deal with the errors have been taken.

```
UF_PART_close_all();
```

Close all loaded parts. This may not be necessary for other programs. We choose to make it a requirement of this program.

```
/* Retrieve calculator.prt using resp = UF_PART_open. Display an error
dialog and return a non-zero if an error occurs. */
strcpy(estr, "./calculator.prt");
resp = UF_CALL( UF_PART_open(estr,&parent,&estat) );
if( resp )
    return( resp );
```

UF_PART_open returns the tag of the part and an error status structure. The structure is declared with `UF_PART_load_status_t estat`. The structure for `UF_PART_load_status` is described in the `uf_part.h` file



```
.  if(estat.n_parts){
    uc1601("Calculator part had components that did not load.",1);
    uc1601("The part should be empty!",1);
    UF_free(estat.statuses);
    UF_free_string_array(estat.n_parts,estat.file_names);
    return( 1 );
}
return (0);
```

Please note that, if there are errors, the memory allocated for the integer array must be freed with *UF_free* and the memory allocated to the string array must be freed with *UF_free_string_array*.



2

Activity: Check Session for Open Part Files

Step 1 Copy existing files *calculator.c*, *calcproto.h*, *calc_part.c*, and *Makefile* to your sub-directory.

Step 2 Edit *calculator.c* and verify the *ufusr_ask_unload* function is present. See page 2–5 for an example. Examine the file *calcproto.h*.

- The prototype include file should have one prototype line. The file will, at a later time, require a typedef declaration made in *uf_modl.h*. The header is included in the template file provided.

```
/* include file calcproto.h */
#include <uf.h>
#include <uf_defs.h>
#include <uf_modl.h>
#include <stdio.h>
/* prototypes for functions in the calculator program */
int calc_part(void);
```

Step 3 Edit *calc_part.c* and add the code necessary to perform the activities stated in the program comments. Variables are already declared for the necessary Open API function calls.

NOTE Comments starting “/*xxx” indicate that a Open API function call should be added. Other comments are included in the program for clarity.



- Step 4** Launch a Unix window with the UG environment by selecting “UGADMIN” from the Uniproducts menu. Pick the “Unix Shell (Start UNIX subshell)” option from the Administration Utilities sub menu. Dismiss the submenu if necessary.
- Step 5** Edit *Makefile* and add the function *calc_part.o* to the SUBOBS area. From the window created in the previous step, use the command “make int” to compile/link your program. You may have to edit your file(s) and fix compile errors.
- Step 6** Create a metric part called *calculator.prt* and save it.
- Step 7** Run the program *calculator*. Try it with parts loaded and without parts loaded.

User Interface Styler

The User Interface Styler lets Unigraphics users and third-party developers generate Unigraphics dialogs. The User Interface Styler is a development tool that:

- Provides a visual builder that allows a developer to correctly build Unigraphics dialogs and generate a User Interface Styler file (with a .dlg extension) that encapsulates the code associated with creating a dialog without requiring comprehensive knowledge of the underlying Graphical User Interface (GUI). The User Interface Styler API insulates the application programmer from the specifics of GUI programming while providing the look and feel of Unigraphics dialogs.
- Reduces development time because of the visual builder and automatic User Interface Styler file generation.
- Allows you to rapidly prototype a dialog by selecting objects such as pushbuttons, toggle buttons, etc. from an object palette list.
- Allows you to select your own user defined bitmaps.
- Provides an attributes editor that allows you to set or modify the attributes of a particular object. For example, the state of a toggle button is an attribute. The attributes editor enables you to specify a valid state for the toggle button (on or off). All of the attributes are conveniently listed when you select an object from a list window.
- Provides context sensitive help for objects located on the User Interface User Interface Styler dialog. The bottom of the dialog window has an area that displays context sensitive information when you position the cursor over an icon.

The User Interface Styler is compatible with UG/Open MenuScript and can be associated with an action in a UG/Open MenuScript “.men” file. Thus, User Interface Styler dialogs can be launched from a MenuScript menubar.

Some Open API functions provide a programmatic interface to the User Interface Styler. These functions will be discussed later in this chapter. The functions are also described in the Open API Reference manual and prototyped in the `uf_styler.h` header file.



Invoking the Styler

You can access the User Interface Styler directly from the Unigraphics Application Pulldown menu (**Application**→**User Interface Styler...**). You are not required to have an open part.

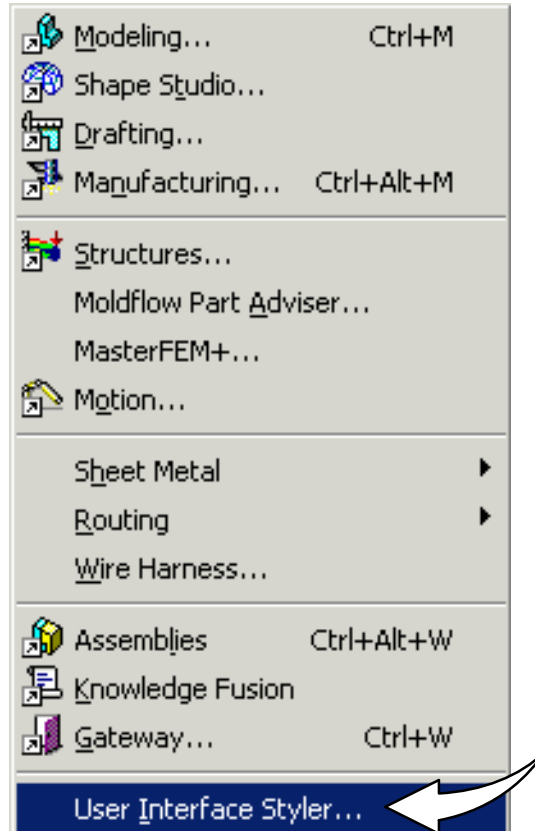
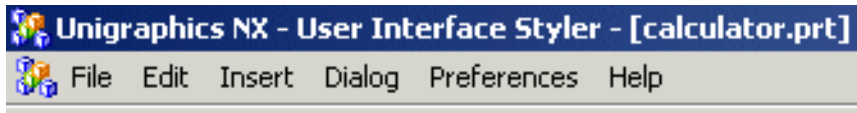


Figure 2–2 Accessing The User Interface Styler

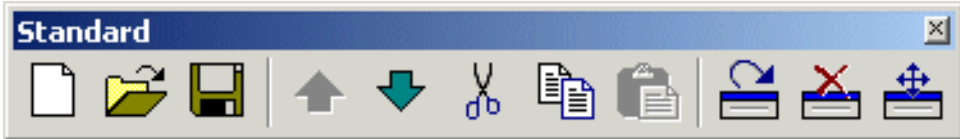
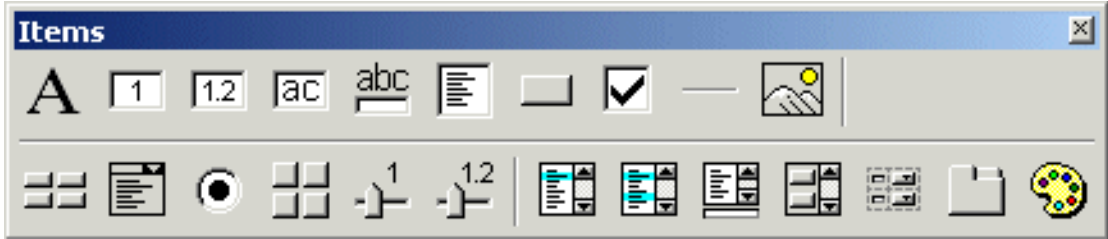
Once you select the User Interface Styler option, the User Interface Styler Graphical Interface dialog and Menubar display along with a default dialog (called the Design Dialog). The Design Dialog displays as an empty dialog box with only the OK, Apply, and Cancel buttons.

Styler Menus

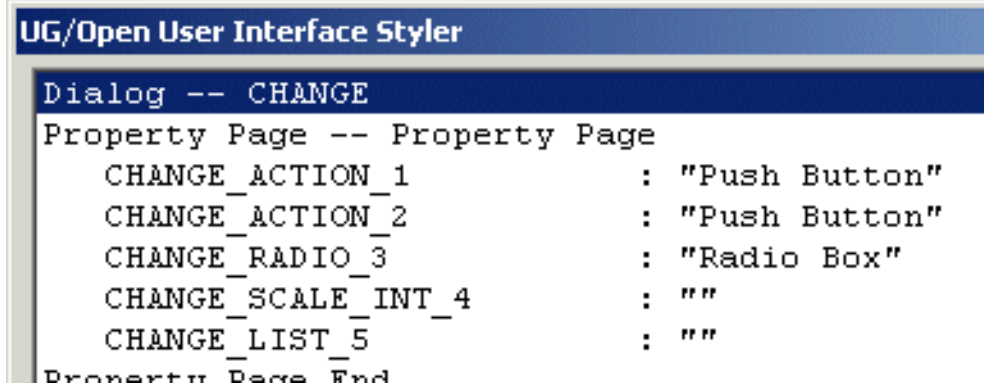
User Interface Styler Menubar



Toolbars

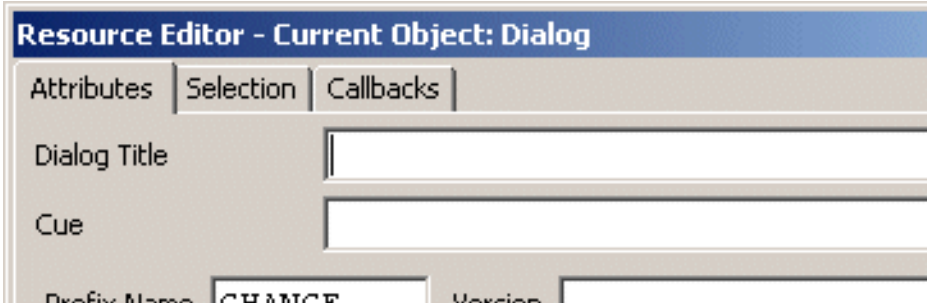


Object Browser




```
Dialog -- CHANGE
Property Page -- Property Page
CHANGE_ACTION_1      : "Push Button"
CHANGE_ACTION_2      : "Push Button"
CHANGE_RADIO_3        : "Radio Box"
CHANGE_SCALE_INT_4    : ""
CHANGE_LIST_5         : ""
Property Page End
```

Resource Editor



Design Dialog



2

User Interface Styler Resource Editor

The large dialog displayed when the User Interface Styler is selected is a focal point for user interaction. Objects created from the toolbar are modified from this dialog. The number of tabbed pages, and the content of those pages, will vary depending on the object selected in the *object browser*.

2

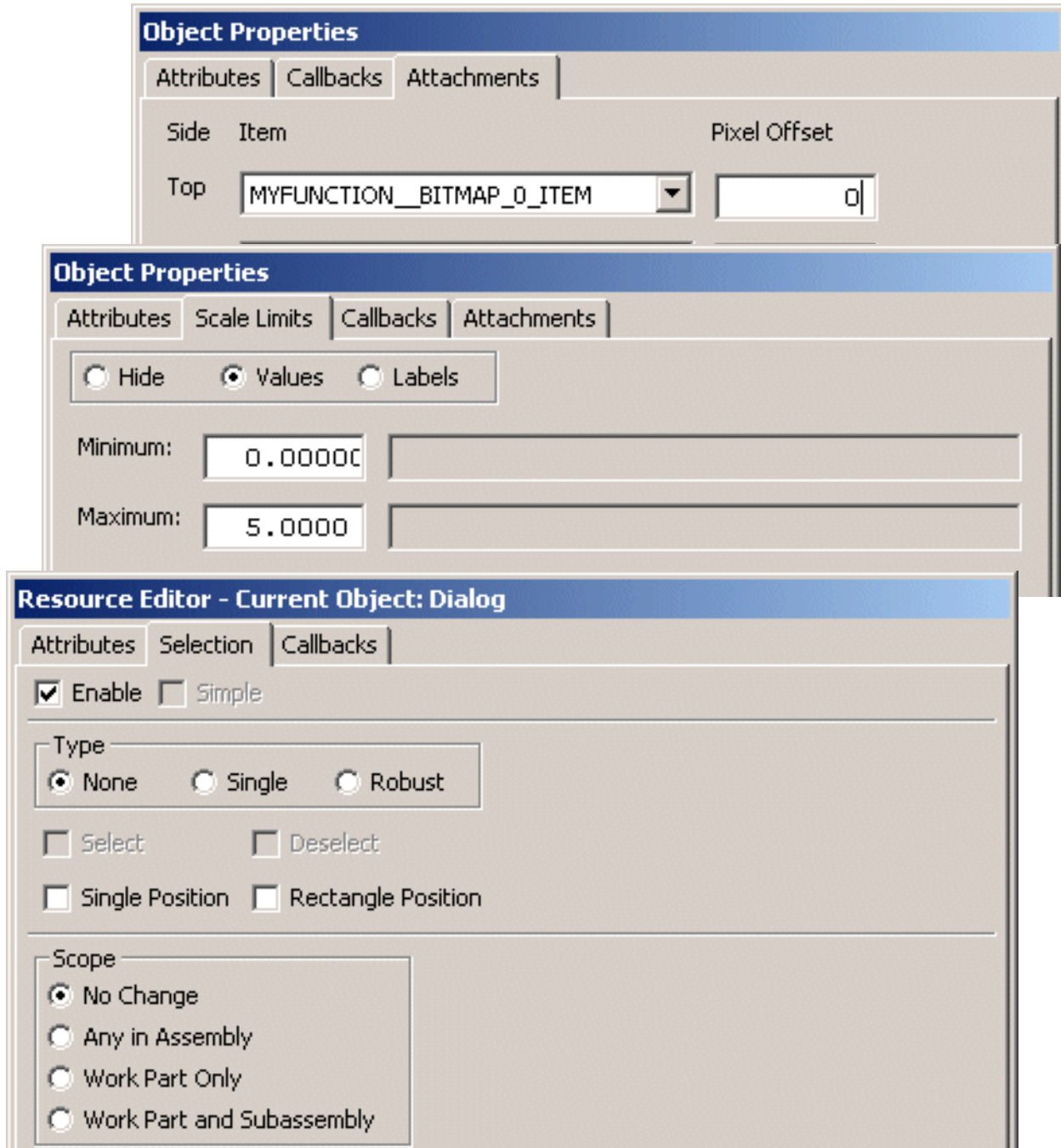


Figure 2–3 User Interface Styler Resource Editor

Object Browser

The Object Browser displays all of the User Interface Objects (UIObjects) that you have constructed in your dialog. The browser allows you to browse through your UIObjects and modify the resources of each object. The Resource Editor automatically updates to display the resources of the item you select in the Object Browser. You may also edit your User Interface Objects with the editing features provided just below the Object Browser.

NOTE The Dialog is also considered a User Interface Object and contains a wide range of helpful resources.

The Object Browser also supports multiple selection of UIObjects in the browser. This allows you to cut/copy/paste groups of UIObjects in your dialog at one time. The Resource Editor is not available when you invoke multiple selection.

Save Dialog

The **Save Dialog** option displays a file selection dialog that lets you specify a pathname to a directory and a name for your dialog. The dialog file name may be up to 30 characters in length, including the extension and any periods. Valid characters for a file name are dependent on the operating system. Pathname specifications can be up to 132 characters in length. If you specify the name of an existing file, an error message displays (Use Save As instead). Enter a file name *without* an extension. Three files are saved to disk:

dialog file	binary file with a “.dlg” file extension.
header file	C header file with a “.h” file extension.
template file	Open API template file with a “.c” extension. The suffix “_template” is appended to the file name that you provide.

The C source will start with many lines of comments to explain the file contents. The comments also include 3 methods of invoking the program. A method can be uncommented or you can create your own call to invoke the dialog. All the callback stubs are created in this source file.

Open API routines for the User Interface Styler

Now that our main program calls our routine to check part status in the Unigraphics session (and retrieve our part), we will add a routine to determine the style, height, and width of the calculator. The routine will present a dialog created using the User Interface Styler. The calculator style determination will be presented in the dialog using a Tool Box. Height and Width will be obtained from real fields. Four fields will be presented; two for horizontal, two for vertical.

The User Interface Styler allows control of the field sensitivity. The height and width will be available (sensitive) for the style of calculator selected. These values will be obtained from the dialog when the user selects the OK button. The Open API provides functions to set and ask about items on a User Interface Styler dialog.

The User Interface Styler can create 2 types of dialogs. A Top dialog acts like a UG Dialog area 1 (e.g. the Modeling palette). A non-top dialog is equivalent to a Dialog area 2 (e.g. the point subfunction menu). This DA2 dialog can have OK, Apply, Back, and Cancel buttons. The API call to create the dialog is automatically generated in the template source file.

- **UF_STYLER_create_dialog**
- **UF_STYLER_ask_value**
- **UF_STYLER_ask_values**
- **UF_STYLER_set_value**
- **UF_STYLER_free_value**

Querying Attributes

The following attributes can be queried via `UF_STYLER_ask_value` or `UF_STYLER_ask_values`:

- **UF_STYLER_SENSITIVITY**
- **UF_STYLER_SELECTION**
- **UF_STYLER_VISIBILITY**
- **UF_STYLER_VALUE**
- **UF_STYLER_SUBITEM_VALUES**
- **UF_STYLER_ITEM_TYPE**

Setting Attributes

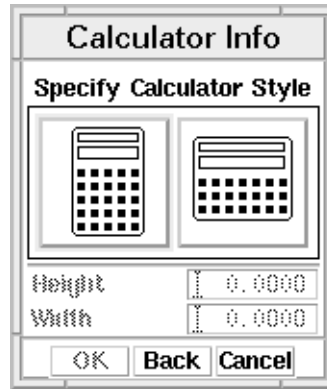
The following attributes can be changed by calling `UF_STYLER_set_value`.

- `UF_STYLER_SENSITIVITY`
- `UF_STYLER_VISIBILITY`
- `UF_STYLER_VALUE`
- `UF_STYLER_LABEL`
- `UF_STYLER_BITMAP`
- `UF_STYLER_FOCUS`
- `UF_STYLER_SUBITEM_VALUES`
- `UF_STYLER_DEFAULT_ACTION`
- `UF_STYLER_DIALOG_WIDTH`
- `UF_STYLER_DIALOG_RESIZE`
- `UF_STYLER_SCALE_PRECISION`
- `UF_STYLER_LIST_UNSELECT`
- `UF_STYLER_LIST_INSERT`
- `UF_STYLER_LIST_DELETE`
- `UF_STYLER_LIST_SHOW:`



Activity: User Interface Styler for Calculator Data

In this activity, you will use the User Interface Styler to create a dialog to prompt for the calculator Style, Height and Width.



Step 1 Obtain the bitmap file from the parts directory.

- Copy the file calculator.ubm to your subdirectory. The file contains two bitmaps shown in the figure above. *Windows users:* see the special instructions later in this activity.

Step 2 Start the User Interface Styler

- Choose **Application** → **User Interface Styler ...**

The Resource Editor and Object Browser are displayed. The Design Dialog appears, with a default configuration consisting of only OK, Apply and Cancel buttons.

Step 3 Specify the resources for the dialog

The default dialog name is CHANGE. This name should be altered to a more meaningful name.

- In the **Prefix Name** field of the Dialog Properties, Attributes page, rename the dialog to **CALC**, and press Enter.

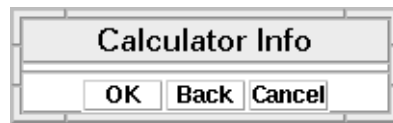
TIP Always hit the Enter key after changing a field in the Dialog Properties Dialog. You cannot harm the dialog by hitting Enter. Data may be ignored if you do not!

- Click in the **Dialog Title** field and type in “Calculator Info” (remember to hit Enter!)
- Click in the **Cue** field and type in “Specify Style, Height, and Width” (remember to hit Enter!)

Leave the **Dialog Type** set to **Bottom**.

- Use the **Button Style Options** drop list to set the buttons to **OK, Back, and Cancel**.

Notice the Design Dialog now contains a Back button instead of the default Apply button.



- Set the **Initially Sensitive** toggle for the OK button to **Off** (not checked.)

The OK button will be unavailable until the user selects a calculator style. We will turn the sensitivity on inside the source code generated for this dialog.

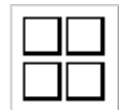
- Set the **Allow Dialog to resize** to **On** (checked.)
- Choose the **Callback** tab.

The **Callback** page presents a list of possible callbacks. These callback functions are bound to buttons/actions by Unigraphics. The Back callback will have a default callback name (because we specified a Back button). The Cancel button will dismiss the dialog without requiring a callback. We will add an OK callback. That function will be augmented with API calls to extract style, height, and width data from the dialog.

- Click in the **OK Callback** field and type in “ok_cb” (remember to hit Enter and do not key in the quotes!)

Step 4 Create buttons to prompt the user for Style.

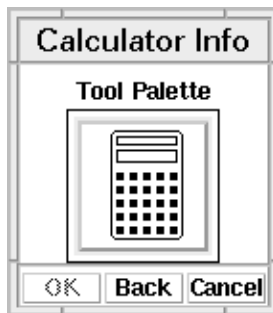
- Choose the **Tool Palette** icon in the **Items** toolbar.



A default Tool Palette will be added to the design dialog, and Object Properties are displayed in the Resource Editor.

Step 5 Specify the resources for the Tool Palette buttons

- In the **Identifier** resource field enter “Style”.
- Click the browse icon beside the **Bitmap** field and navigate to your calculator.ubm.



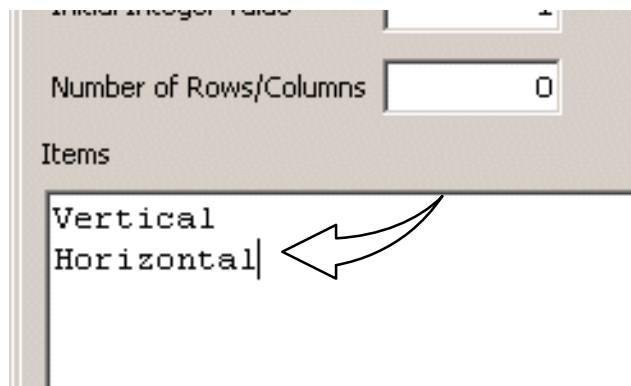
NOTE

Windows Users – the UBM bitmap format is Unix only. Use calculator_win.ubm instead of calculator.ubm. This is a simple text file. You must edit this file to give the *full and complete path name* for two bitmap files in your files directory, calculator1.bmp and calculator2.bmp.

The Initial Integer Value field should be left a –1. This indicates that *no* style is selected initially. When the vertical style is selected, the value will become 0. For horizontal, the value would be 1.

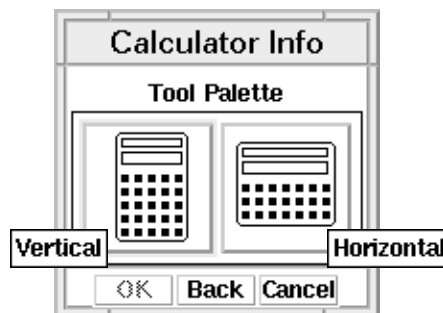
The ubm file can contain multiple bitmaps. The Items field controls the number of icons (buttons) that will be presented.

- Key in the words “Vertical” and “Horizontal” on separate lines in the **Items** list window.



- Choose **Apply** in the **Resource Editor** dialog.

Two buttons will appear on the dialog. If you place the cursor over a button, the name on the Item list will be displayed.

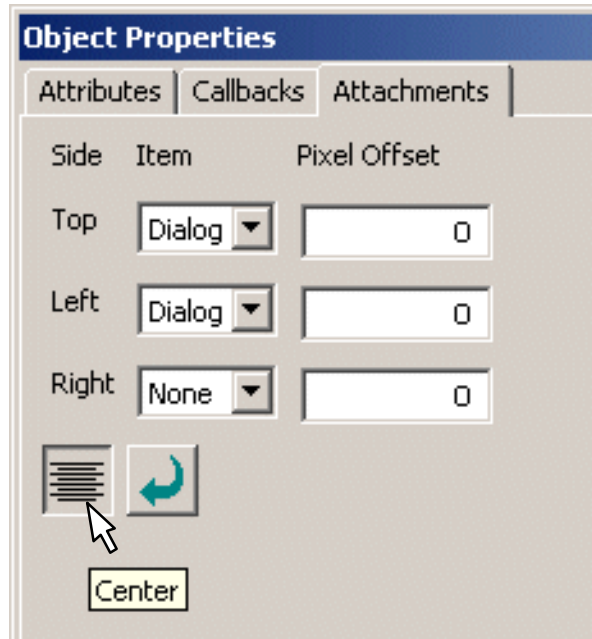


- In the **Label** field enter “Specify Calculator Style.”
- Choose the **Callbacks** tab.
- In the **Activate** window enter “style_cb.” Remember to press Enter.

Step 6 Edit the Tool Palette attachments to center the icons.

- Choose the **Attachments** tab.

- Make sure the **Center** toggle is **ON** in the **Attachments** page.



You will not see any immediate difference in the Design Dialog. This step insures the tool palette will remain centered when more items are added to the dialog.

Step 7 Allow the user to specify Height/Width

- Select the **Separator** icon. 

The separator attachments can be edited at any time. You may want to distance the separator line from the icons.

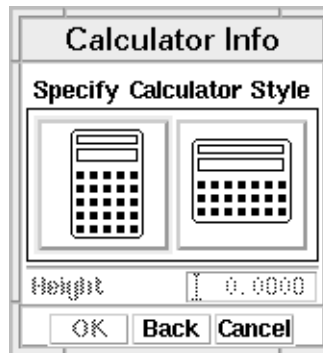
- Select the **Real** icon. 

- In the **Identifier** property field key in “Height” and press enter.

We’ll leave the initial real value at 0. When the user picks a style, we will load the default values for that style.

- In the **Label** property field key in “Height” and press enter.
- Change the **Sensitive** toggle to **False**; that is, set the check box to **not** checked.

This field will be inactive (not sensitive) until a style is chosen. We will not specify an Activate Callback.



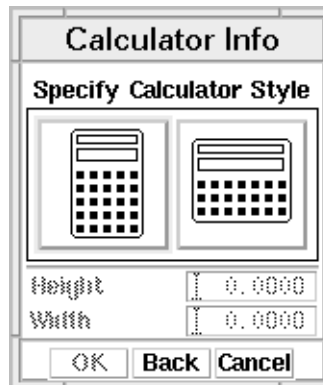
- Once again, select the **Real** icon. 

- In the **Identifier** property field key in “Width” and press enter.

We’ll leave the initial real value at 0. just as we did for height.

- In the **Label** property field key in “Width” and press enter.
- Change the **Sensitive** toggle to **False**; that is, set the check box to **not** checked.

This field will also be inactive until a style is chosen.



The dialog will be saved in its current configuration. Changes to the dialog attachments (changing the arrangement of objects) should be done *after* the exercise is complete.

Step 8 Save the dialog, source, and header files.

- Choose **File** → **Save**.
- Make sure you are in your subdirectory! Enter the file name “xxx_calc_setup” (where xxx are your initials).

You will receive a message from UG indicating that all User InterfaceStyler files have been saved. The Dialog, Include, and Template file names will be listed in the message.

Step 9 Exit the Styler Application.

- Choose **File** → **Exit Styler**
- Choose “Yes” to indicate that you really wish to quit.

Step 10 Adjust the source file name and dialog file directory.

The Styler will overwrite the template file, dialog file, and header file every time you save. Since you want to edit the template file, you will rename it to prevent overwrites.

The dialog file should be located in an application directory. UG automatically looks in specific directories for these files.

Unix commands are shown. Windows users are familiar with the corresponding Windows actions.

- Rename the template file from a Unix Shell by typing in the command (do this from your working directory):

```
mv ../xxx_calc_setup_template.c xxx_calc_setup.c
```

- Move the dialog file to the application directory (from a Unix Shell) by typing in the command:

```
cd ~/application
mv ../xxx_calc_setup.dlg xxx_calc_setup.dlg
```



Step 11 Edit the header file and include our prototype.

- Edit the file `xxx_calc_setup.h` and add the line

```
#include <uf_styler.h>
#include "calcproto.h"
```

Step 12 Edit the source file and create an entry point.

The source file created for the dialog has numerous comment statements. We will uncomment one of the commented sections for invoking the dialog.

- Find the section of code that is set up for creation from a callback. Look for the following comment:

```
/*-----DIALOG CREATION FROM A
CALLBACK HELP-Example-----
If you wish to have this dialog displayed from the callback of
another User Interface Styler dialog, you should:
```

- Delete or comment the `#ifdef` line. The statement appears once in the comment statements in the program. The second occurrence starts in column one and appears as:

```
#ifdef DISPLAY_FROM_CALLBACK
```

- Enter a function name. Our function will pass back height, width, and style. The return value will be nonzero for Back, Cancel or any error condition.

```
extern int <enter the name of your function> ( int *response
)
{
```

Becomes...

```
extern int calc_setup( int *style, double *height, double
*width )
```

- Remove only the Initialize and Terminate calls

```
if ( ( error_code = UF_initialize() ) != 0 )
    return (0) ;
.
.
.
UF_terminate();
```



- Add an integer variable called response. Also create a local structure to use as client data in the dialog creation call. The client data pointer is passed into all callback functions so each callback can change data in the structure. You will set initial values in the structure as well. Global variables could also be used.

```
int error_code = 0, response;
struct calc_data_s
{   int style;
    double height;
    double width;
} calc_data = {-1, 0., 0.};
```

- Change the dialog create call to pass the address of the structure rather than a NULL pointer. Change response to pass by address.

```
if ( ( error_code = UF_STYLER_create_dialog (
"xxx_calc_setup.dlg",
    CALC_cbs, /* Callbacks from dialog */
    CALC_CB_COUNT, /* number of callbacks*/
    &calc_data, /* This is your client data */
    &response ) ) != 0 )
```

- Extract the style, height and width from the structure and return a success code (a zero) if OK was selected. Return non-zero if Back or Cancel was selected.

```
return (error_code);
```

becomes:

```
if( UF_UI_OK == response ) {
    *style = calc_data.style;
    *height = calc_data.height;
    *width = calc_data.width;
    return( 0 );
}

return( 1 );
}
```

- Remove the #endif statement

```
#endif /* DISPLAY_FROM_CALLBACK */
```

Step 13 Add function calls to the callbacks.

We want to edit the callbacks and add the code to read the values. We will declare a pointer to our structure in each callback. The style variable will be set in the style callback. The height/width will be obtained in the OK callback

- Find the callback section of code.

```
/*-----*/
/*----- UIStyler Callback Functions -----*/
/*-----*/
```



- Create a pointer to the calc_data structure. Declare an array of two dialog item value structures. Declare an int called 'count'.

```
int CALC_ok_cb ( int dialog_id,
                void * client_data,
                UF_STYLER_item_value_type_p_t callback_data)
{
    struct calc_data_s
    { int style;
      double height;
      double width;
    } *calc_data_p;

    UF_STYLER_item_value_type_t dlg[2];

    int count;
```

- Remove only the UF_initialize and UF_terminate calls.

```
/* Make sure User Function is available. */
if ( UF_initialize() != 0)
    return ( UF_UI_CB_CONTINUE_DIALOG );
    .
    .
    .
UF_terminate ();
```

- Add a call to `UF_STYLER_ask_values` to access the height and width. Prior to calling the API function, you must set the item id and item attribute for both dialog item structures. You must also set the structure pointer `calc_data_p` to the client data pointer.

```
/* ----- Enter your callback code here ----- */
calc_data_p = client_data;
```

```
dlg[0].item_id = CALC_HEIGHT;
dlg[0].item_attr = UF_STYLER_VALUE;
```

```
dlg[1].item_id = CALC_WIDTH;
dlg[1].item_attr = UF_STYLER_VALUE;
```

```
UF_CALL( UF_STYLER_ask_values( dialog_id, 2, dlg,
&count ) );
```

```
calc_data_p->height = dlg[0].value.real;
calc_data_p->width = dlg[1].value.real;
```

Next, the Back callback will be changed. There is no activity associated with this call but the initialize and terminate calls should be eliminated.

- Remove the `UF_initialize` and `UF_terminate` calls from the `calc_back_cb` function.

Code from
calc_back_cb

```
/* Make sure User Function is available. */
if ( UF_initialize() != 0)
    return ( UF_UI_CB_CONTINUE_DIALOG );
.
.
.
UF_terminate ();
```

The final callback is the associated with the tool palette for style. The initialize and terminate calls should be eliminated first. The style value will be obtained here. A function has been provided to make the OK button active and to change the data in the Height/Width fields. You will add a call to this provided function.

- Remove the `UF_initialize` and `UF_terminate` calls from the `calc_style_cb` function.

Code from
calc_style_cb


```

/* Make sure User Function is available. */
if ( UF_initialize() != 0)
    return ( UF_UI_CB_CONTINUE_DIALOG );
    .
    .
    .
UF_terminate ();

```

- Create a pointer to the `calc_data` structure. Declare a dialog item value structure.

```

int CALC_style_cb ( int dialog_id,
                  void * client_data,
                  UF_STYLER_item_value_type_p_t callback_data)
{
    struct calc_data_s
    { int style;
      double height;
      double width;
    } *calc_data_p;

```

UF_STYLER_item_value_type_t dlg;

- Extract the style information from the dialog. You must also set the structure pointer `calc_data_p` to the client data pointer.

The toolbox palette identifies the item selected through the `UF_STYLER_VALUE` attribute. The integer value is the zero based index of the item selected. Our `calc_data_p->style` variable will be set to 1 for vertical, 2 for horizontal. This value is the index (dialog) value +1.

```

/* ----- Enter your callback code here ----- */
calc_data_p = client_data;

```

```

dlg.item_id = CALC_STYLE;
dlg.item_attr = UF_STYLER_VALUE;

```

```

UF_CALL( UF_STYLER_ask_value( dialog_id, &dlg ) );

```

- Call the routine that activates OK, sets the height/width values. Only call the routine if the style has changed.



```
/* This callback can be invoked when the user picks on
 * a style that is already selected. If the style has
 * not changed, don't set the calc_data_p value or set
 * the style dialog.
 */
if( calc_data_p->style != dlg.value.integer + 1 ) {
    calc_data_p->style = dlg.value.integer + 1;
    calc_set_style_dialog( dialog_id, calc_data_p->style );
}
```

- Save your file. You have created a function named *calc_setup* in a file named *xxx_calc_setup.c*. You will call the setup function from *calculator*.

Step 14 Change other associated source and header files. Update the Makefile.

- Edit the function *calc_set_style_dialog.c* and change the include file name. It will be *xxx_calc_setup.h*. Replace the xxx with your initials.
- Edit the *calculator.c* program and add a call to *calc_setup*. Remember to pass the style, height, and width by address. Also remember that your source file name is different than your function name!
- Edit *calcproto.h* to include prototypes for *calc_setup* and *calc_set_style_dialog*. (Called by *calc_setup*)
- Edit the Makefile and add the functions *xxx_calc_setup.o* and *calc_set_style_dialog.o* to the SUBOBJS list. Use “make int” to try compiling/linking your program.



UI Routines for Object Selection

Open API programs allow users to select geometric objects from the current graphics window. Routines to mask (set a filter of selectable objects) and prompt for single/multiple object selection are provided.

The following are defined in the header file `uf_ui.h`. The scope parameter is used to control the selection in assemblies. The parameter must be specified for both simple parts and assemblies.

- `UF_UI_SEL_SCOPE_WORK_PART` – Allows you to select only objects which belong to the work part. This includes immediate components of the work part. If you select an object occurrence, the prototype is returned.
- `UF_UI_SEL_SCOPE_ANY_IN_ASSEMBLY` – Allows you to select any object or object occurrence in the assembly. No scope restrictions are applied.
- `UF_UI_SEL_SCOPE_WORK_PART_AND_OCC` – Allows you to select objects which belong to the work part or its subassembly. If you select an object occurrence, the prototype is returned.

Object selection is controlled by either a list of objects types or through a “mask triple” structure. The mask triples requires three values (instead of a single number for object type) to allow access to detail filtering. The header file `uf_object_types.h` lists the valid objects types for simple filtering used in `UF_UI_set_select_mask()`. Mask triples are used in `UF_UI_set_sel_mask()`.

- `UF_UI_set_select_mask`
- `UF_UI_select_with_class_dialog`
- `UF_UI_select_with_single_dialog`
- `UF_UI_set_sel_mask`
- `UF_DISP_set_highlight`
- `UF_DISP_set_display`
- `UF_DISP_add_item_to_display`
- `UF_DISP_regenerate_display`
- `UF_DISP_refresh`

SUMMARY

You used the legacy UI routines to prompt the user about open files. You closed and perhaps saved open parts before opening our calculator part. You also used the User Interface Styler to create a custom dialog to prompt for calculator style, height, and width.

In this lesson, you :

- Learned how to open/close and save part files.
- Learned about memory management for Open API functions in Unigraphics.
- Learned about the capabilities in the User Interface Styler API routines, as well as the User Interface Styler User Interface Dialog.

Assemblies

Lesson 3

An understanding of the part data model in assemblies is important whether or not you currently use Unigraphics Assemblies and Components. This lesson will introduce students to the intricacies of assemblies and the Open API functions provided to work with the assembly model.

OBJECTIVES

Upon completion of this lesson, you will be able to:

- Learn the structure of the Unigraphics Assemblies and Components data model.
- Understand the relationship between Instances, Occurrences, and Prototypes.
- Become familiar with the function used to create an assembly structure.
- Obtain information about object tags, types, and subtypes.



The handheld calculator to be created will consist of the following parts:

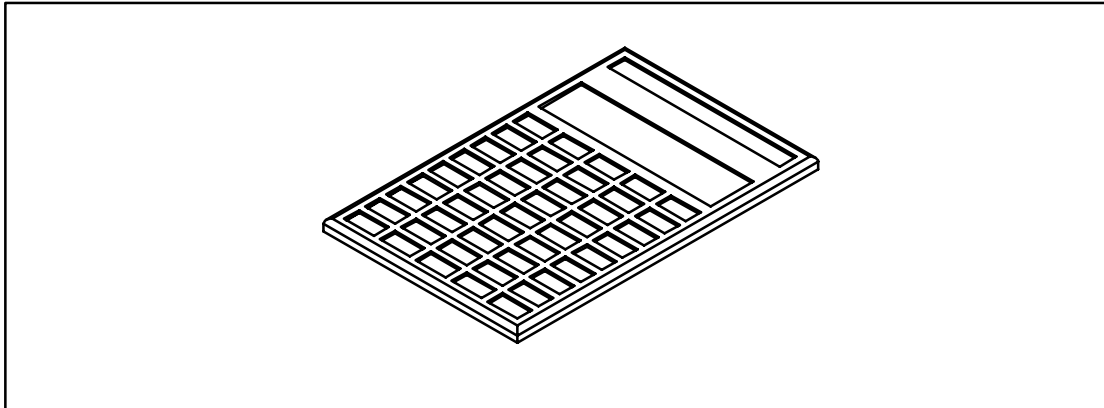


Figure 3-1 The top half of the unit.

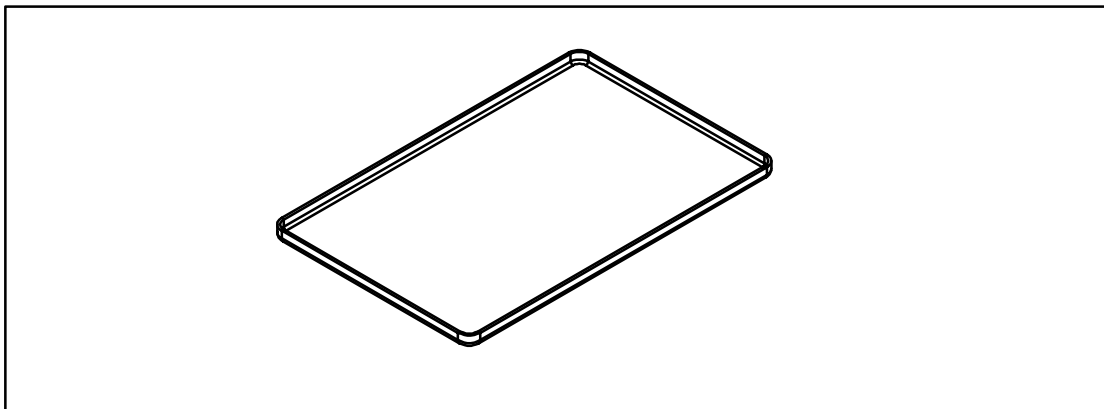


Figure 3-2 The bottom half of the unit.

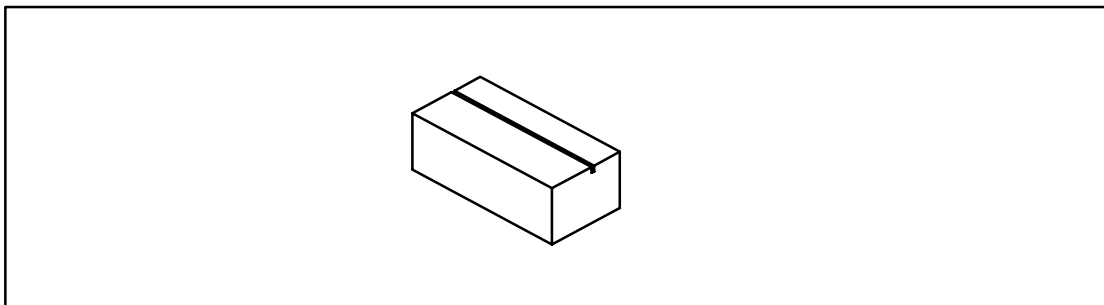


Figure 3-3 The individual button.

3

Top–Down Design

The program will use the top-down and design-in-context approaches to assembly modeling. It will create a component for each of the listed items. The component for the buttons will be a single button for which we will create a master component instance set array later in the program.

In order for us to use the top-down and design-in-context approaches, we'll first create the assembly structure. Then for the construction of each portion (top, bottom, buttons), we'll change the work part to be the component part. We will also change layers as we create the components so each will be on a separate layer in the assembly part.

Assembly Terminology and Concepts

The following are important terms used when discussing assemblies:

A **tag** identifies an object in the data model. The objects are not limited to geometrical objects, but can include parts, instances, and occurrences.

A **Piece Part** is the lowest level part in an assembly. It contains model geometry and feature information that represent the part, but no component objects.

When several parts are brought together into an assembly, the parts and their objects are not copied into the assembly part. Rather, the parts are loaded into memory, and **part occurrences** of those parts are put into the assembly part. For each object in the piece part, an **object occurrence** of that object is made in the assembly part.

An **Instance** is the term used to indicate the placement of a component part within an assembly part. For each component instance, a part occurrence is displayed in the assembly part.

Assemblies can be multi–leveled. For example, an automobile can consist of the body and two instances of an axle assembly part, which itself consists of an axle and two instances of a wheel assembly part, which itself consists of other parts. Each instance can have more than one associated part occurrence.

A **Component Part** is any part used at least once in an assembly. A component may be a sub-assembly consisting of other, lower-level components. Each component object in an assembly contains only a link to its master geometry. When you modify the geometry of one component, all other components in the session using the same master will automatically update to reflect the change.



Newer (Unigraphics V10 and after) style components are listed as Type *UF_component_type*, Subtype *UF_part_occurrence_subtype*. Rarely, you may encounter old components, Unigraphics V9 and before. These are stored in the Data Model as Type *UF_component_type*, Subtype *UF_component_subtype*.

A **Reference Set** is a named collection or set of geometry from a Unigraphics part. The Reference Set may be used to simplify the representation or display of the component part in larger or complex assemblies.

The **Displayed Part** is the part that is being viewed in the Unigraphics graphics window. It can be a piece part or an assembly.

The **Work Part** is the part whose geometry or assembly structure is being modified. The Work Part can be the same as the Displayed Part. When the display part is an assembly, the work part can be any of the component parts in the assembly. When the Displayed Part and the Work Part are different, making modifications to the work part is termed **Designing in Context**.

A **Prototype** is a master copy of a part or object *occurrence*. As mentioned in the Component Part topic, occurrences are links to master or prototype geometry. To edit an object, you must make the Prototype part the Work Part and edit the prototype object.

Sample Assembly

Using a simplistic example of an automobile, we can view it in two ways: a graph and a tree. The logical graph highlights the “Instances:”

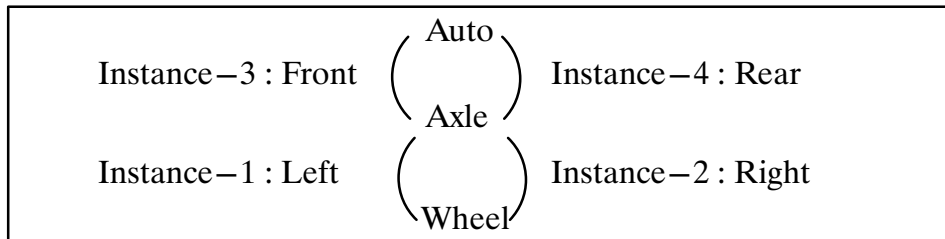


Figure 3-4 Logical Graph

From the graph, we get the following occurrence trees:

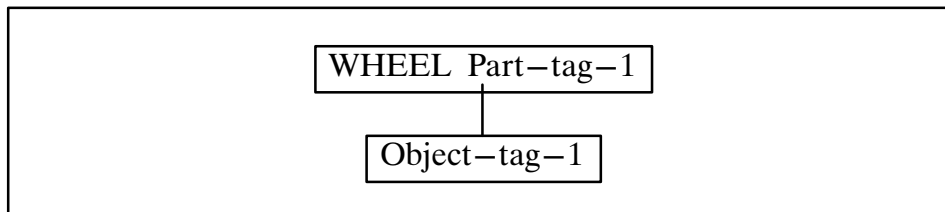


Figure 3-5 WHEEL.PRT



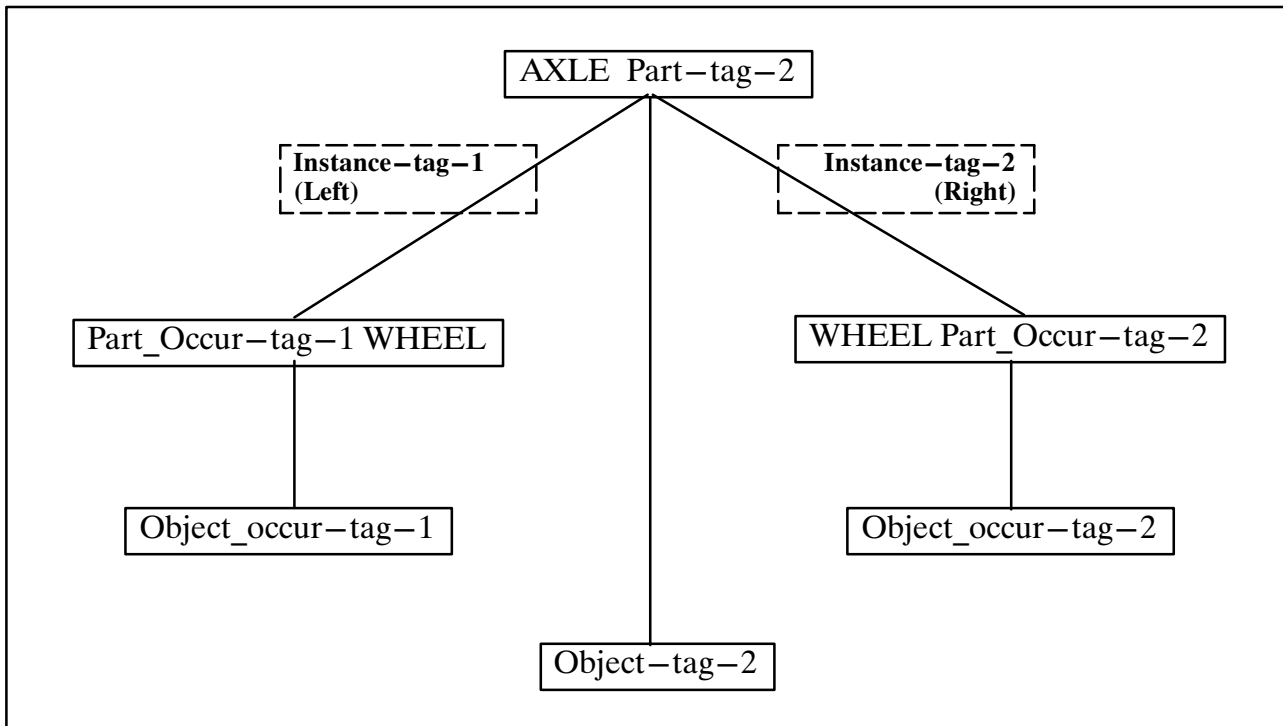
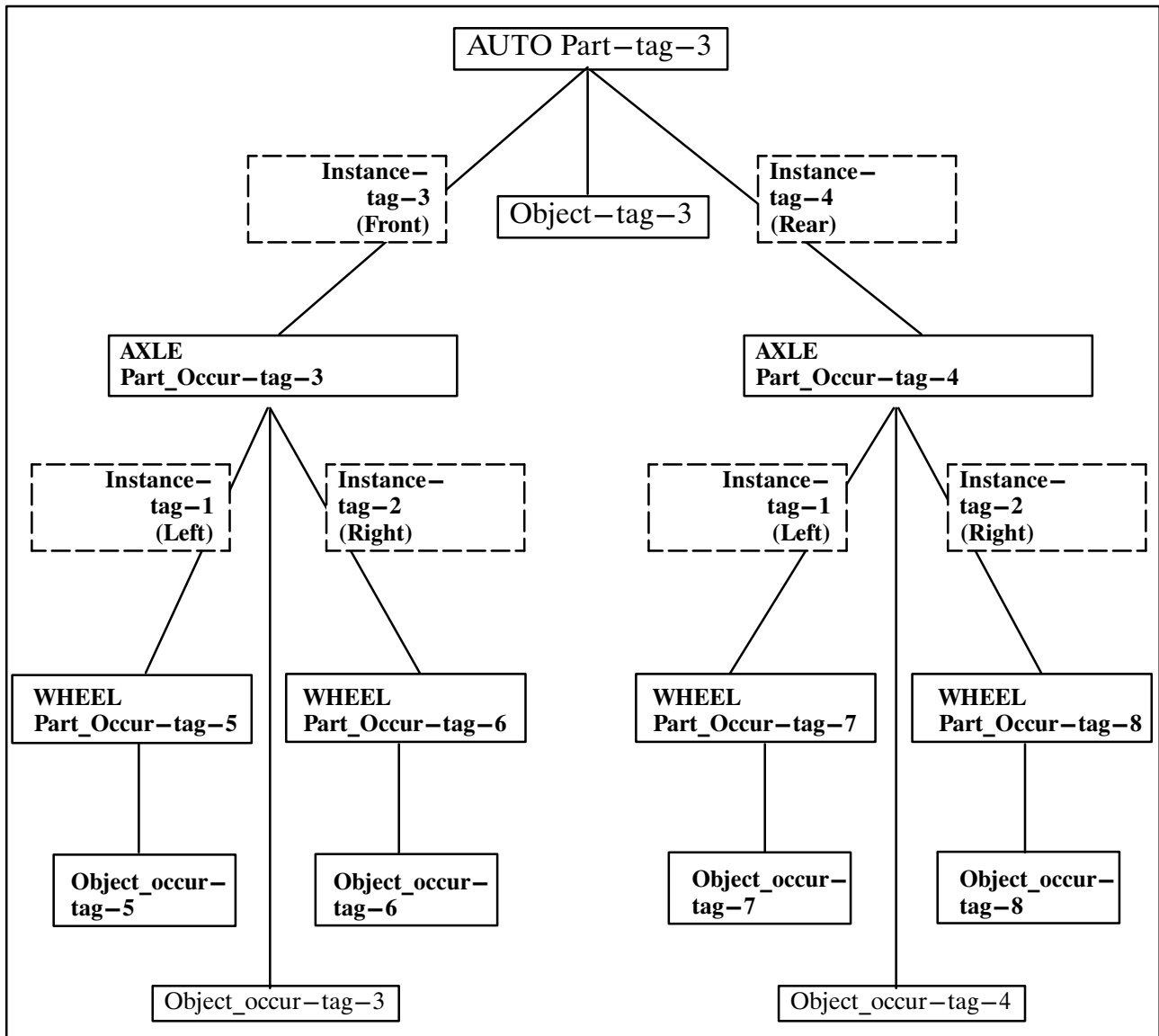


Figure 3-6 AXLE.PRT

3



3

Figure 3-7 AUTO.PRT

The charts emphasize “Part Occurrences”.

Explanation:

The part WHEEL is loaded and given the Part tag Part-tag-1. It consists of a single object Object-tag-1.

The part AXLE is loaded and given the Part tag Part-tag-2. It consists of an object Object-tag-2 for the axle and two instances of the WHEEL piece part. Instance-tag-1 represents the Left wheel and Instance-tag-2 represents the Right wheel. Two Part occurrences are created for these and given the tags Part_Occur-tag-1 and Part_Occur-tag-2. Similarly, two Object Occurrences are created for the wheel object, one for each Part Occurrence, and given the tags Object_occur-tag-1 and Object_occur-tag-2.

The part AUTO is loaded as the Displayed Part and given the Part tag Part-tag-3. It consists of an object Object-tag-3 and two instances of the AXLE assembly part. Instance-tag-3 represents the Front axle, and Instance-tag-4 represents the Rear axle.

Since there are two instances of the AXLE part in the AUTO assembly part, two Part Occurrences are created for them and given the tags Part_Occur-tag-3 and Part_Occur-tag-4. Similarly, two Object Occurrences are created for the axle object, 1 for each Part Occurrence, and given the tags Object_occur-tag-3 and Object_occur-tag-4.

Since there are two instances of the wheel part in the AXLE assembly part, two Part Occurrences are created for them for every occurrence of AXLE in the session. Thus, 4 more Part Occurrences are created for WHEEL. There is a Part Occurrence for the Left wheel Instance (Instance-tag-1) of the Front axle Instance (Instance-tag-3) called Part_Occur-tag-5 and a Part Occurrence for the Right wheel Instance (Instance-tag-1) of the Front axle Instance (Instance-tag-3) called Part_Occur-tag-6. There is also a Part Occurrence for the Left wheel Instance (Instance-tag-1) of the Rear axle Instance (Instance-tag-4) called Part_Occur-tag-7 and a Part Occurrence for the Right wheel Instance (Instance-tag-1) of the Rear axle Instance (Instance-tag-4) called Part_Occur-tag-8. Similarly, 4 more Object Occurrences are created for the wheel object, one for each Part Occurrence of WHEEL. They are named Object_occur-tag-5, Object_occur-tag-6, Object_occur-tag-7, and Object_occur-tag-8.

The following table indicates the routines to access given a particular tag when you want to find a tag of a different type. For example, if you have a “PART tag” and need the “PART name” you would call *UF_PART_ask_part_name*.

TO FIND	GIVEN	CALL
PART name	PART tag	UF_PART_ask_part_name()
PART tag	PART name PART OCCUR tag INSTANCE tag OBJECT OCCUR tag	UF_PART_ask_part_tag() UF_ASSEM_ask_prototype_of_occ() UF_ASSEM_ask_parent_of_instance() UF_ASSEM_ask_child_of_instance() 1) UF_ASSEM_ask_part_occurrence() + 2) UF_ASSEM_ask_prototype_of_occ()
PART OCCUR tag	PART tag PART OCCUR tag INSTANCE tag OBJECT OCCUR tag	UF_ASSEM_ask_occs_of_part() UF_ASSEM_ask_part_occ_children() UF_ASSEM_where_is_part_used() UF_ASSEM_ask_part_occs_of_inst() UF_ASSEM_ask_part_occ_of_inst() UF_ASSEM_ask_part_occurrence()
INSTANCE tag	PART tag PART OCCUR tag INSTANCE name OBJECT OCCUR tag	UF_ASSEM_cycle_inst_of_part() UF_ASSEM_ask_inst_of_part_occ() UF_ASSEM_ask_instance_of_name() 1) UF_ASSEM_ask_part_occurrence() + 2) UF_ASSEM_ask_inst_of_part_occ()
OBJECT tag	OBJECT OCCUR tag OBJECT handle	UF_ASSEM_ask_prototype_of_occ() UF_TAG_ask_tag_of_handle()
OBJECT OCCUR tag	PART OCCUR tag OBJECT tag OBJECT handle	UF_ASSEM_cycle_ents_in_part_occ() UF_ASSEM_find_occurrence() UF_ASSEM_ask_occs_of_entity() UF_TAG_ask_tag_of_handle()
OBJECT handle	OBJECT tag	UF_TAG_ask_handle_of_tag()
Misc		UF_ASSEM_is_occurrence() UF_ASSEM_ask_transform_of_occ() UF_ASSEM_part_is_descendant() UF_ASSEM_count_ents_in_part_occ()

Reference Sets – Whenever the name of a reference set is used in this document as a parameter to a routine, if the string is NULL/blank, the whole part is used. If the string “Empty” is used, the empty reference set will be used.

Displayed Part – Any fully loaded part in memory can be made the Displayed Part. A partially loaded part can be made the displayed part by opening it.

Work Part – Any fully or partially loaded part which is a member of the assembly under the Displayed Part can be made the Work Part. Only the objects of the Work Part may be edited. Object Occurrences can NOT be edited, but they may be queried to find their data. Thus, the Object Occurrence of one of the wheel occurrences in AUTO may be read to find what it’s center is, but the Object Occurrence can not be edited. If the actual object (prototype) in the WHEEL part is edited, all occurrences of that object will be updated to show the modification.

Coordinate Systems

All coordinate system references consist of an array of 6 floating point numbers which correspond to two unit vectors. The first vector gives the direction of the X axis, while the second gives the approximate direction of the Y axis. The routine *UF_MTX3_initialize* will always be called to make sure the vectors are ortho-normal.

When an instance is added to an assembly or moved in an assembly, an Origin and Matrix are specified. The Origin is the position in the work part where the absolute origin of the component part or the origin of the component reference set is placed. The instance will be transformed so that its absolute CSYS or the CSYS of its reference set (if specified) is matched to this Matrix at the Origin point.

Assembly Routines

- **UF_ASSEM_create_component_part**
- **UF_ASSEM_create_mc_array**
- **UF_ASSEM_ask_work_part**
- **UF_ASSEM_ask_component_data**
- **UF_ASSEM_ask_mc_array_data**
- **UF_ASSEM_ask_assem_options**
- **UF_ASSEM_is_occurrence**
- **UF_ASSEM_is_part_occurrence**
- **UF_ASSEM_ask_prototype_of_occ**
- **UF_ASSEM_ask_inst_of_part_occ**
- **UF_ASSEM_ask_parent_of_instance**
- **UF_ASSEM_ask_child_of_instance**
- **UF_ASSEM_remove_instance**
- **UF_ASSEM_ask_occs_of_entity**
- **UF_ASSEM_ask_occs_of_part**
- **UF_ASSEM_set_assem_options**
- **UF_ASSEM_cycle_ents_in_part_occ**



Layer Routines

The following Open API routines are used to control and read the layer settings for the work part.

- **UF_LAYER_set_status**
- **UF_LAYER_ask_status**
- **UF_LAYER_cycle_by_layer**

Object Routines

The following Open API routines relate to objects. There are other Open API object routines pertaining to object names in a later lesson.

- **UF_OBJ_delete_object**
- **UF_OBJ_ask_status**

NOTE You are strongly advised to avoid doing anything to non–alive objects unless you are familiar with their use. UG may delete or reuse these objects at any time. Some of these objects do not get filed with the part during a save operation.

- **UF_OBJ_set_def_cre_color**
- **UF_OBJ_set_cre_color**
- **UF_OBJ_ask_type_and_subtype**

NOTE The types and subtypes this function returns for expressions and parts are not useful in any other Open API routines. We provide these types only to allow you to determine the class of a Unigraphics object through its identifier (i.e. its tag) which can assist you in determining the other Open API routines that you can use in conjunction with the object. For a more detailed discussion of classes of objects and the Open API routines available to these objects, please see the Open API Programmer's Guide, "The Unigraphics Object Model".



Code Discussion: *calc_assem.c*

```
char
  pname[MAX_FSPEC_SIZE+1],      /* new part name */
  refset[MAX_ENTITY_NAME_SIZE+1], /* reference set name */
  iname[MAX_ENTITY_NAME_SIZE+1]; /* name of comp inst */
```

`MAX_FSPEC_SIZE` and `MAX_ENTITY_NAME_SIZE` are defined in *uf_defs.h*. They define the maximum count of characters allowed for file specifications and entity names. One extra character is added to account for the nul character as a string terminator in C.

Check *uf_defs.h* to find what other maximum sizes are defined and use the definitions in your own projects. The same issues discussed in the last chapter with typedefs apply here. If you use the `#defines`, your code will be more maintainable. If the value of either of these macros changes, your code will still work.

```
const int
  zero=0, one=1, two=2, three=3;
```

It is often convenient to place each component on its own layer. We're going to put our components on layers one through three.

TIP Variables or literals can be used for constant values passed as arguments. A variable, however can be passed by address for use in legacy routine calls. Using `const int one=1` is equivalent to `#define ONE ((int) 1)`.

```
double
  imat[]={1.0, 0.0, 0.0,
          0.0, 1.0, 0.0 }, /* orientation for components */
  origin[]={0.0, 0.0, 0.0}; /* origin for each component */
```

For those of you who are familiar with matrices and transformations, the *imat* variable is the first two vectors of an identity matrix. Although you might expect a matrix to have nine positions, most matrices in Unigraphics are unitized and orthogonal. Thus, the third column is the cross product of the first two. Since the last three values can be inferred from the first six, only the first six are used in *UF_ASSEM_create_component_part*.

The origin variable is the offset portion (translation) associated with the component. As geometry is created in each component, it is positioned in relationship to that component's coordinate system. For creating a calculator, we have chosen to set all the components' coordinate systems to the absolute coordinate system. We can apply all positioning to the geometry itself rather than using mating conditions.

```
tag_t  
    nullt = NULL_TAG;
```

UF_ASSEM_create_component_part accepts an array of tags of geometry to be transferred from the assembly to the new component. As we are going to create our own geometry, we'll pass the address of a `tag_t`. Rather than pass an uninitialized variable, we set it to `NULL_TAG`.

```
strcpy(pname, "./top_half");  
refset[0] = '\\0';  
strcpy(iname, "TOP_HALF");
```

The part name is lower case because the system part files will be in lower case. The relative directory path is used here but a full path can also be used. The instance name (`iname`) can be specified in lower case but will be converted to upper case by Unigraphics. Here we specify it in upper case to highlight that fact.



TIP The part name, reference set name, and component name are arguments to *UF_ASSEM_create_component_part*. But problems can occur if they are passed in as string literals. String literals should be used only when a variable is declared as `const char`. It may be that Unigraphics tries to add “.prt” which would overwrite memory. Declare character arrays with the `MAX` size definitions for these variables and use *strcpy* to load them.

Creating a Component

```
flag = UF_ASSEM_create_component_part (workp, pname, refset, iname,
                                       units, one, origin, imat,
                                       zero, &nullt, top_inst_p);
```

Create a component:

- Make it child of the work part: (variable *workp*)
- Name the part file in the file system: (*strcpy(pname, "top_half")*)
- Do not bother to define a reference set: (*refset[0] = '\0'*)
- Name the component object: (*strcpy(iname, "TOP_HALF")*)
- Set the construction units to millimeters: (variable *units*)
- Place the component object on layer 1: (variable *one*)
- Set the component origin to the absolute CSYS: (array *origin*)
- Set the component orientation to that of the absolute CSYS: (array *imat*)
- Do not add geometry to the component yet: (variables *zero* objects, *&nullt* no object tags being passed)
- The function will return a tag for the new instance (tag *top_inst_p*)

We are going to start with the top–down design method where the components are created without any geometry. Then, using the design–in–context method, each component, one at a time, will be the work part and geometry will be created within it at that time.

Activity: Create Assembly Parts for Calculator

Our next task is to define and create the component part files. This is done by the function *calc_assem*.

- Step 1** Copy the template file for *calc_assem.c* to your sub-directory.
- Step 2** Edit the program and add code to set the active layers using *UF_LAYER_set_status* and create empty component parts using *UF_ASSEM_create_component_part*.
- Step 3** Edit the header file *calcproto.h* and add the *calc_assem* prototype.
- Step 4** Edit the file *calculator.c* and add the call to *calc_assem*. Note that *calc_assem* returns a success/failure status. This status must be evaluated and a message displayed.
- Step 5** Edit *Makefile* and add *calc_assem.o* to the SUBOBJS list and make a new executable. Use the Assemblies Navigation Tool (ANT) to view the structure you just created.

There is no geometry yet but, after the program runs, select **Assemblies**→**Reports**→**List Components** or **View**→**Assembly Navigator**, to see that the components have been created.



Report Object Tags, Types, and Subtypes

- Step 1** Copy existing file *show_tags.c* to your sub-directory.
- Step 2** Edit *show_tags.c* and verify the *ufusr_ask_unload* function is present.
- Step 3** Add the code necessary to perform the activities stated in the program comments. Variables are already declared for most of the necessary UG/Open API function calls.

NOTE Comments starting “/*xxx” indicate that a UG/Open API function call should be added. Other comments are included in the program for clarity.

- Step 4** Create a make file for the program. Name the file *select* because you already have a file called *Makefile* in your directory. Use “make ext -f select” to create your executable.
- Step 5** Open the part called *test_show_tags*.
- Step 6** Run the program *show_tags*. Select objects and observe their tag and other information reported in the information window.
- Step 7** Optionally, create some new objects and run the program again.
- Step 8** Close the part without saving.

SUMMARY

You created an empty assembly using the Top–down approach. The master model calculator now has three component parts, `top_half`, `bottom_half`, and `button`.

In this lesson, you :

- Used some of the assembly routines to create a small assembly.
- Performed an introductory activity to obtain information about object tags, types, and subtypes.

(This Page Intentionally Left Blank)



Expressions

Lesson 4

Expressions in Unigraphics are arithmetical or conditional statements that can be used to control the characteristics of a part. Expressions can define many dimensional values of a model. For example, expressions are used for the scalar values or positioning dimensions of a feature or dimensional constraints of a sketch.

OBJECTIVES

Upon completion of this lesson, you will be able to:

- Create and modify expression strings.
- Create a conditional expression.



Expression Definition

An expression is a statement defining a relationship. An *expression relation* is both the left–hand side and right–hand side of an expression equation (i.e. $a=b+c$). All expressions have a value (a number which may or may not have a fractional part) which is assigned to the variable on the left side of the expression. To obtain this value, the system evaluates the right side of the expression, which is either a mathematical or conditional statement. The left side of the expression must always be a single variable. If the expression name is to be used as a variable in another expression (on the right side), it must be defined prior to its usage.

Expression names are alphanumeric strings of text, but they must begin with a letter. An underscore, “_”, may also be used as a “word delimiter” within an expression name for clarity (no embedded spaces). Expression text is treated by the expression system inside of Unigraphics as case sensitive, so the variable name “X1” is different from “x1”.

Conditional Expressions

You may create conditional expressions by using the if/else structure. The if/else expression uses the following syntax:

$$\text{VAR} = \text{if} (\text{expr}_1) (\text{expr}_2) \text{ else } (\text{expr}_3)$$

For example,

$$\text{width} = \text{if} (\text{length} < 8) (2) \text{ else } (3)$$

means that:

if length is less than 8, width will be 2;

if length is greater than or equal to 8, width will be 3.

A full description of expressions can be found in the Documentation under Design→Modeling. The documentation describes the interactive creation of expressions, the operators (addition/subtraction, etc.), and the built in functions (e.g., `abs()`) available.

The calculator we are going to build is going to have a top half that is five millimeters thick and a bottom half that is five millimeters thick. Thus, the entire thickness will be ten millimeters.

All heights and widths will be calculated from the height and width that the user specifies (from *calc_setup*). Expressions will be used to store the initial values of the calculator height and width. Once the calculator is modeled, the size may be changed easily by the user simply by interactively changing the height and width expressions.

Now that the assembly structure has been set up, it is time to create the model geometry in the component part files. The first thing to do is to create a fundamental set of expressions that will control the model feature geometry. After that, editing the expressions will cause changes to the appropriate component model geometry.



Expression Routines

The following routines can be used to create and manipulate expressions from the Open API:

- **UF_MODL_create_exp**
- **UF_MODL_ask_exp**
- **UF_MODL_edit_exp**
- **UF_MODL_delete_exp**
- **UF_MODL_eval_exp**
- **UF_MODL_export_exp**
- **UF_MODL_import_exp**
- **UF_MODL_rename_exp**
- **UF_MODL_create_exp_tag**
- **UF_MODL_ask_exp_tag_value**
- **UF_MODL_delete_exp_tag**
- **UF_MODL_ask_exp_tag_string**
- **UF_MODL_dissect_exp_string**
- **UF_MODL_update**

Expressions for the calculator

The calculators will have the following expressions, as shown in the following figures. The hole is going to be a little larger than the button (for clearance purposes). The “outside” buttons are positioned using the Border expression.

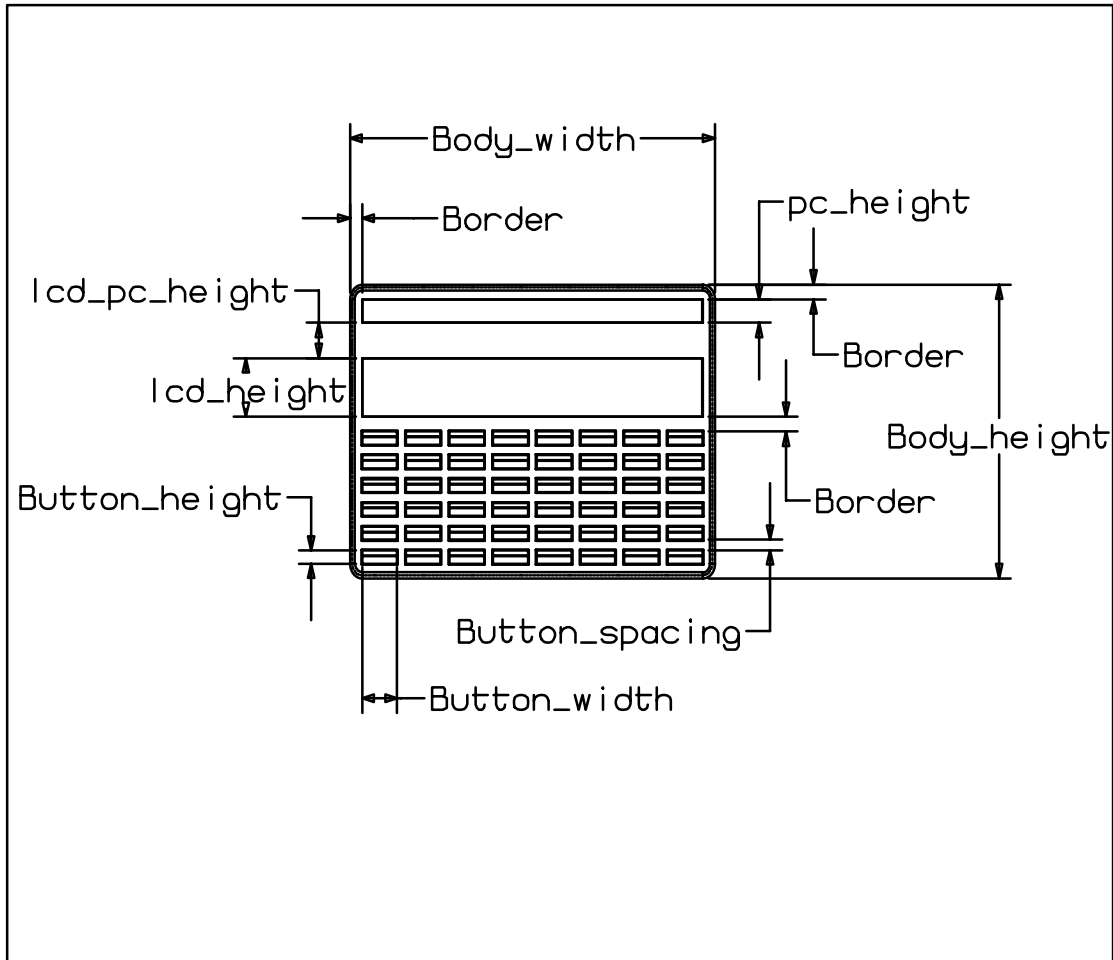


Figure 4–1 Horizontal Calculator

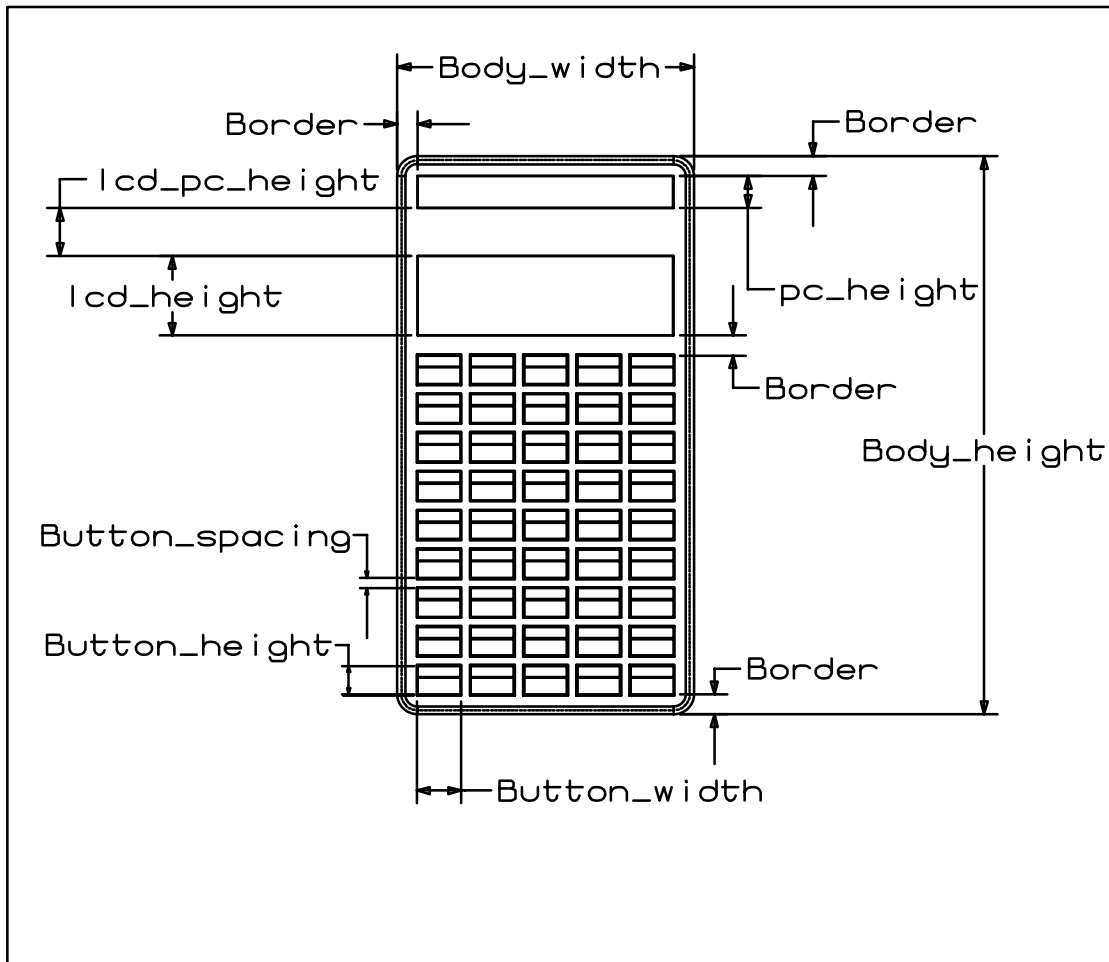


Figure 4-2 Vertical Calculator

Use the previous figures to develop the logic for the calculator expression creation program. The `Border` expression will be a fixed value of 5 mm. The photo cell height (`pc_height`) will be a fixed value of 8 mm. The liquid crystal display height (`lcd_height`) will be 20 mm. The distance between the display and photocell (`lcd_pc_height`) will be a fixed value of 12 mm. The values of `Body_width` and `Body_height` will be the data obtained from the `calc_setup` routine. The `Button_height` and `Button_width` expression will be driven by the overall body height and width.

The `Button_width` expression will vary with the style of calculator. The spacing will be .25 of the `Button_width`. The vertical calculator has 5 buttons across and 4 spaces. The 4 spaces, at .25 of the width per space, are equivalent to 1 button. For a vertical calculator, the expression for `(width/#buttons)` is:

$$\text{Button_width} = (\text{Body_width} - 2.0 * \text{Border}) / (5.0 + 1.0)$$

Similarly, the width for the horizontal calculator must accommodate 8 buttons and 7 spaces (at .25 of the width per space). The 7 spaces are equivalent to 1.75 buttons. For the horizontal calculator ($\text{width}/\#\text{buttons}$) is:

$$\text{Button_width} = (\text{Body_width} - 2.0 * \text{Border}) / (8.0 + 1.75)$$

The denominators, of course, can be simplified to 6.0 and 9.75 for the vertical and horizontal calculators, respectively.

An intermediate expression, Bh1, will be used to keep the string length of the Button_height expression less than 132 characters. Bh1 is the entire height of the calculator minus the heights for the photocell and lcd windows and the space between them. Three times the border size is subtracted to account for the top, bottom, and space between the lcd and the button array.

Now, if the style is vertical ($\text{Style}=1$), the buttons are nine high. There are 8 spaces to be considered for the space between the buttons and each space is 25% of the button width. Thus, the button height is the length defined by Bh1 minus the spacing between the buttons divided by the count of buttons. Arithmetically, it looks like this:

$$\text{Button_height} = (\text{Bh1} - (9.-1.) * 0.25 * \text{Button_width}) / 9.$$

For the horizontal calculator, there are 6 buttons high. It's height would be:

$$\text{Button_height} = (\text{Bh1} - (6.-1.) * 0.25 * \text{Button_width}) / 6.$$

In actual usage, $(9.0-1.0)*0.25$ should be shortened to 2.0 and $(6.0-1.0)*0.25$ should be shortened to 1.25.

It is important that Button_height is defined after Button_width and Bh1. Remember, if an expression's definition depends upon other expressions, those expressions must already exist before the new expression is created.



Code Discussion: calc_expr.c

The routine will set up expressions based on the style of calculator selected.

```
/* Create expression called Style. Set its value to the function arg.
 * 'style'. Check the return flag and exit if it is nonzero.
 */
    sprintf(estr, "Style=%d", style);
```

The line `sprintf(estr, "Style=%d", style);` translates the integer value of `style` to a string prefixed by the name `style=`. The character pointer `estr` should point to an array large enough to contain the translated string. The expression is either "Style=1" or "Style=2". The expressions for

- Height
- Width
- `lcd_height` (liquid crystal display window height)
- `pc_height` (photocell window height)
- `lcd_pc_height` (vertical distance between liquid crystal display and photocells)
- Border

are also straightforward and follow `Style` in the function.

```
strcpy(estr, "Across=if (Style==1) (5) else (8)");
flag = UF_CALL( UF_MODL_create_exp(estr) );
if(flag) return (flag);
```

Across is going to be used for the buttons and their holes. It depends upon `Style` and therefore must be defined after the `Style` expression has been registered (created). What this expression says is "If `Style` is equal to 1, `Across` is equal to 5. If `Style` is not equal to 1, `Across` is equal to 8".



Activity: Create `calc_expr`

Step 1 Copy the template file `calc_expr.c` to your sub-directory.

Step 2 Create the expressions for the calculator creation based on the style of calculator.

Step 3 Change the `calculator` program to call the expression creation routine.

Step 4 Add the prototype to the header file `calcproto.h`.

```
int calc_expr(int, double, double);
```

Step 5 Edit `Makefile` and add `calc_expr.o` to the `SUBOBJS` list and make a new executable. Use the expression browser under `Info` → `Expression` → `List All` to verify the expressions have been created.

SUMMARY

You created the expressions necessary to control the calculator expression. The expressions were created in the calculator part.

In this lesson, you :

- Generated expressions using data from the custom dialog.
- Generated expressions using constant values.
- Created conditional expressions.

Modeling

Lesson 5

This lesson discusses the construction of the model geometry and the manipulation of the components. Geometry construction will be accomplished through feature–based modeling. The Open API Modeling routines allow for the creation, editing and inquiry of objects in the part database.

OBJECTIVES

Upon completion of this lesson, you will be able to:

- Create primitive and form features.
- Understand the relationship between Body, Feature, Face and Edge.
- Use the linked list of tags as output or input to modeling functions.
- Create/remove names from objects.
- Cycle the data model to find objects based on names.



Geometry Creation in Context

Before creating the model geometry for the calculator, we must control the context of the model geometry creation. Geometry creation will be done in each component individually. The assembly we have created has an assembly part file and three component parts. The calculator part (assembly) is the current work and display part. To create geometry in a component part, we must change the work part to the component part, then create the appropriate component model object(s).

Each component part is positioned in the assembly using an instance. We obtained the instance tag when creating the component parts. The instance positions a *child* part relative to a *parent* part. The geometry can be created in the appropriate part file by making each instance's child part the work part.

Model list routines

The model routines that create or provide object/feature tags currently use the `uf_list_p_t` structure to provide data. These routines could be obsoleted in the future in favor of arrays. The following routines show how to create a `uf_list_p_t` structure, populate the list, and deal with the linked lists.

These routines are defined in `uf_modl_utilities.h` and documented under `uf_modl_general`:

- **UF_MODL_create_list**
- **UF_MODL_put_list_item**
- **UF_MODL_ask_list_count**
- **UF_MODL_ask_list_item**
- **UF_MODL_ask_list_item**
- **UF_MODL_delete_list**
- **UF_MODL_delete_list_item**

Code Discussion: calc_model.c

```
/* Set the variable 'parent' to the current work part */
parent = UF_ASSEM_ask_work_part();
```

Save the current work part tag (the root of the assembly tree) so we can restore the work part. We will be changing the work part to each component part to create geometry and should return the work part to its original value when completed.

In order to set the display and work part, we need to get the tag of the part. *UF_ASSEM_ask_child_of_instance* gets the tag of the part from the associated component object.

```
/* Obtain the top piece part tag (wpart) using ask_child_of_instance
 * (check the return tag validity). XXX Change the display part to the
 * top piece part. Check return code)
 */
wpart = UF_ASSEM_ask_child_of_instance(top_inst);
if(NULL_TAG == wpart) return (1);

flag = UF_CALL( UF_PART_set_display_part(wpart) );
if(flag) return (2);
```

There is a *UF_ASSEM_set_work_part* function but we don't need it because *UF_PART_set_display_part*, discussed in a previous chapter, sets the work part to the new displayed part.

TIP Whether *UF_PART_set_display_part* sets the work part is controlled by assembly options. If you want to make sure, use *UF_ASSEM_set_assem_options* in *calc_assem*.

```
/* Create the interpart expressions for the work part. Check return
 * code. Use UF_MODL_update after creating the expressions.
 */
flag = calc_interpart_expr();
if( flag ) return(flag);
flag = UF_CALL(UF_MODL_update() );
if( flag ) return(flag);
```

In order for us to construct our model, we first create the expressions that the model depends upon. Interpart expressions, or IPE, is a powerful, fully associative, optional approach to controlling geometry in an assembly. The interpart expressions need to be created in each of the piece parts, by sequentially making them the displayed and work part.

```
/* Set the solid body creation color to white */
UF_CALL( UF_OBJ_set_cre_color(
    UF_solid_type,
    UF_all_subtype,
    UF_OBJ_solid_body_property,
    UF_OBJ_WHITE ));
```

Sixteen basic color values, and the current maximum number of colors, are defined in *uf_obj_types.h*.

Object types and subtypes are defined in *uf_object_types.h*.

Object properties are also defined in *uf_obj.h*. The property is currently `UF_OBJ_no_property` for everything *except* solids. Solids have either `UF_OBJ_solid_body_property` or `UF_OBJ_sheet_body_property`.

In the documentation for *uf_obj.h* near the bottom of the page, above *uf_obj_errors.h*, there is a link to information about the proper usage of type, subtype, and properties. The parameters above were chosen with guidance from that page.

calc_model sets the default object creation color to various values. Notice that the function used is specific to the work part.

Geometry will be differently colored in each component, making solids visually easier to identify.

After setting a color, changing display and work part, and creating expressions in each part, there is a call to a function that actually creates geometry. At this time we will leave all such function calls commented out. We will uncomment the calls as each function is defined.

Activity: Create Routine `calc_model`

Step 1 Copy the template file `calc_model.c` to your sub-directory. Edit the program and add the appropriate Open API calls.

NOTE Leave the calls to `calc_model_top`, `calc_model_bottom`, `calc_model_button`, and `calc_comp_array` commented out for now.

Step 2 Add a call to `calc_model` in your `calculator` program.

Step 3 `calc_model` requires a `uf_list_p_t` as an input parameter. Add a call to `UF_MODL_create_list`, using the address of `holes`, a variable that was declared earlier.

Step 4 Add a call to free (delete) the list at the corresponding comment later in `calculator.c`.

Step 5 Edit `calcproto.h` and add the prototype for the new routines.

```
int calc_interpart_expr(void);  
int calc_model(tag_t, tag_t, tag_t, uf_list_p_t);
```

Step 6 Edit the `Makefile` and add `calc_model.o` to the `SUBOBJS` list and make a new executable.

To check the expressions, use **Info**→**Expression**→**List All in Assembly**.

Do not save the part files created!

Calculator Top Half

The top half of the calculator is made from a hollowed block with rectangular pockets and blended edges. The modeling routines to create and manipulate these and other features are provided. The Open API Reference documentation contains the complete list of modeling routines under various headings such as `uf_modl` and `uf_modl_features`. The routines are prefixed by `UF_MODL_`. Prototypes and `#define` variables are found in the header file `uf_modl.h` and `uf_object_types.h`.

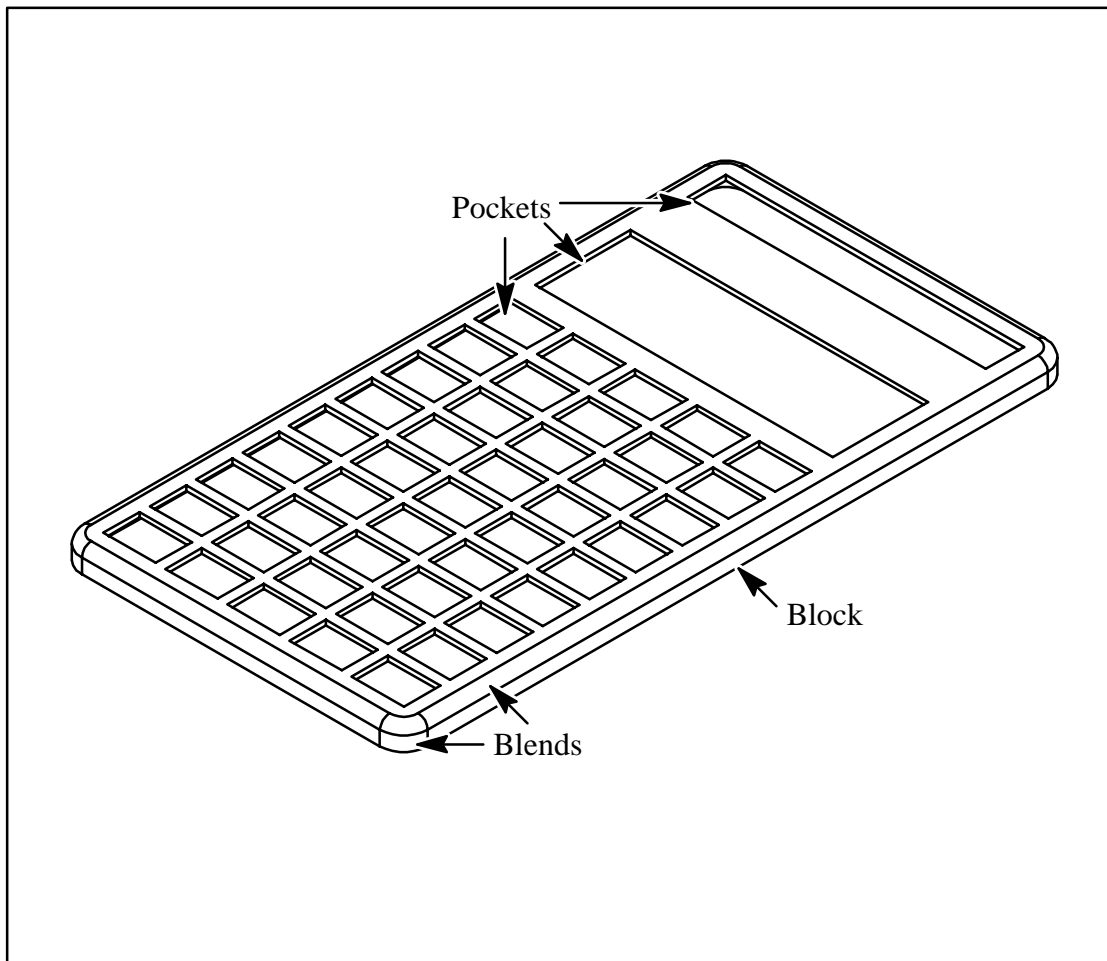


Figure 5–1 The top half of the unit

The dimensions for many of the `UF_MODL_create` routines use character strings for data. This allows the input of a complete expression (e.g. “`bl_hgt=3.125`”) or just the rhs (e.g. “`3.125`”, “`dia`”). Unigraphics will create a ‘p’ number expression when just the right–hand side (rhs) is used (defined).

Several inquire routines with the prefix “UF_MODL_ask_” return lists of items. Since identifiers exist for bodies, features, faces and edges, care must be taken to use the correct identifiers as input into other “UF_MODL_xxxx” routines. The routines that return bodies, faces and edges contain identifiers that can be used throughout the Open API. But the routines that return feature identifiers can be used almost solely within other Open API functions with the prefix “UF_MODL_” that require a feature for input.

For example, calling UF_MODL_ask_body_feats() returns a list of features that are connected to the input body identifier. The identifiers contained in this list are valid in the following routines only:

```
UF_MODL_ask_xxxx_parms()
UF_MODL_ask_feat_xxxx()
UF_MODL_move_feature()
UF_MODL_delete_feature()
```

A common mistake in using the UF_MODL_xxxx routines involves the hierarchy of the solids created. The descending order is body, feature, face, and then edge. When you create the first primitive, you must realize that the identifier returned is **not** a body identifier. To get the body, you must first call UF_MODL_ask_feat_body() to get the proper body identifier.

The “UF_MODL_ask_” routines that inquire about bodies, faces, edges, etc. do not guarantee a consistent order with respect to location in a list. For example, if you get the faces on the block, a specific face is not always found at the same location in the list. The “UF_MODL_ask_feat_” routines do return features in a consistent order.

Target Solid for an Operation

When you have more than one solid in a part and you wish to perform an operation on a solid such as adding a feature or doing a boolean, you must identify which solid is the target for the operation. Use the function UF_MODL_active_part to identify the target solid. You do not need to identify a solid if it is the only solid or if it was the last solid you worked on interactively.

Model Creation Routines

The following subset of routines demonstrate feature model creation capabilities available in the Open API (see `uf_modl_features`):

- **UF_MODL_create_block1**
- **UF_MODL_create_blend**
- **UF_MODL_create_cyl1**
- **UF_MODL_create_hollow**
- **UF_MODL_create_linear_iset**
- **UF_MODL_create_rect_pocket**
- **UF_MODL_create_rect_slot**
- **UF_MODL_create_simple_hole**
- **UF_MODL_operations**
- **UF_MODL_active_part**

Model inquiry routines

The following subset of routines provide information about features and objects Unigraphics in the Unigraphics Part file. (see uf_modl)

- **UF_MODL_ask_feat_body**
- **UF_MODL_ask_body_type**
- **UF_MODL_ask_body_faces**
- **UF_MODL_ask_body_edges**
- **UF_MODL_ask_edge_body**
- **UF_MODL_ask_edge_faces**
- **UF_MODL_ask_edge_type**
- **UF_MODL_ask_edge_verts**
- **UF_MODL_ask_face_body**
- **UF_MODL_ask_face_edges**
- **UF_MODL_ask_face_data**
- **UF_MODL_ask_feat_faces**
- **UF_MODL_ask_block_parms**
- **UF_MODL_ask_minimum_dist**
- **UF_MODL_ask_simple_hole_parms**

Object Name Routines

Unigraphics allows geometric and group objects to be named. Names can be a maximum of 30 characters. The names can be used in interactive selection and cycling of the data model. (see uf_obj)

- **UF_OBJ_set_name**
- **UF_OBJ_ask_name**
- **UF_OBJ_delete_name**
- **UF_OBJ_cycle_by_name**

Code Discussion: *calc_model_top.c*

```
edge_len_p[0] = &edge_len[0][0];
edge_len_p[1] = &edge_len[1][0];
edge_len_p[2] = &edge_len[2][0];

flag = UF_CALL( UF_MODL_create_block1(UF_NULLSIGN, corner,
                                     edge_len_p, &top_half) );
```

This is the function that creates a block. We will use `UF_NULLSIGN` to just create a feature rather than performing a boolean with another object.

Unigraphics allows you to designate one solid as a target solid so that boolean operations can be applied as each tool is created. See *UF_MODL_active_part*, under `uf_modl`.

TIP Note that the edge lengths are input as strings rather than numbers. This enables the programmer to use expressions for the edge lengths. As you can see from the code preceding this line, we are taking advantage of this capability.

```

/* Obtain the body tag from the feature tag. Assign the name
 * TOP_HALF to the body. Check the return code.
 */
flag = UF_CALL( UF_MODL_ask_feat_body(top_half,&body) );
flag = UF_CALL( UF_OBJ_set_name(body,"TOP_HALF" ) );

```

The tag returned by *UF_MODL_create_block1* is a tag to a feature. The first line of code gets the body of a feature. The body object, not the feature, should be named. The second line of code assigns a name to the solid body.

```
flag = UF_CALL( UF_MODL_ask_body_edges(body,&list1) );
```

Now we fill our list with the edges in the body.

We want to blend the four short edges that go from the back of the block to the front. In order to do this, we have to identify those edges. We'll do so by creating a list of all the edges in the body and then finding which edges are five millimeters long.

The Open API provides list handling facilities. With them, you can put items on a list, remove items from a list, ask a list how many items it has, and sequentially process the items on a list.

NOTE The list structure will be obsoleted in favor of arrays of tags.

```
flag = UF_CALL( UF_MODL_ask_list_count(list1,&count) );
```

We're going to use a loop to inquire about each edge. This function tells us how many edges we're going to loop through.

```
flag = UF_CALL( UF_MODL_create_list(&list2) );
```

When we find an edge to be blended, we're going to put it on this second list. *UF_MODL_create_list* creates a list and returns its pointer. We didn't need to create **list1** because it is created and allocated by the routine that returns the list. (In this case, *UF_MODL_ask_body_edges*.)

```

for(i=0; i < ecount; i++) {
/* Get the edge (list item) and check return code. */
    flag = UF_CALL( UF_MODL_ask_list_item(list1,i,&edge) );

/* Get the edge vertices. Check return code. */
    flag = UF_CALL( UF_MODL_ask_edge_verts(edge,pt1,pt2,&vcount) );
    if(flag) {
        UF_CALL( UF_MODL_delete_list( &list1 ) );
        UF_CALL( UF_MODL_delete_list( &list2 ) );
        return (flag);
    }

/* Put item on list2. Check return code. */
    if(fabs(fabs(pt1[2] - pt2[2]) - 5.0) < 0.001) {
        flag = UF_CALL( UF_MODL_put_list_item(list2,edge) );
        if(flag) {
            UF_CALL( UF_MODL_delete_list( &list1 ) );
            UF_CALL( UF_MODL_delete_list( &list2 ) );
            return (flag);
        }
    }
}
}

```

This loop checks every edge in **list1**. If the edge is five millimeters long in Z, it is added to **list2**.

```

flag = UF_CALL (UF_MODL_create_blend("5.0",list2,allow_smooth,
    allow_cliff,allow_notch,vrb_tol,&blend1) );

```

Here, we create a five millimeter radius blend on the edges in **list2**. Note that the radius is input as a string. We could have used a complex expression so that the blend would change with the calculator height or width.

```

flag = UF_CALL( UF_MODL_delete_list(&list1) );
if(flag) {
    UF_CALL( UF_MODL_delete_list( &list2 ) );
    return (flag);
}

```

```

flag = UF_CALL( UF_MODL_delete_list(&list2) );
if(flag) return (flag);

```

The next body of code blends the edges of the top face. In preparation, **list1** and **list2** are deleted. **list1** is refilled and **list2** is recreated. We refill **list1** because the previous blending operation created new edges. The edges to be blended are those whose start and end vertices both have a Z coordinate of 10.0.

After the edges are blended, **list1** and **list2** are reset again. **list1** is then filled with the faces of the body and **list2** contains the face whose normal has a Z component of -1.0 .

```
flag = UF_CALL( UF_MODL_create_hollow("1.0",list2,&hollow) );
```

Hollow out the face(s) in **list2**. Although the remaining shell is coded to have a thickness of one millimeter, the thickness could have been defined by a more complex expression.

```
/* Create the holes (pockets) */  
/*      flag = calc_model_top_holes(body,holes);*/  
      if(flag != 0) return (flag);
```

The basic body of the top half of the calculator has been created. *calc_model_top_holes* will create the holes (as rectangular pockets) for the windows and the buttons. **The routine will be uncommented in another exercise.**

The *calc_model_top* routine, before the call to *calc_model_top_holes*, generates a hollowed, blended shape that looks like the following (from a TOP view):

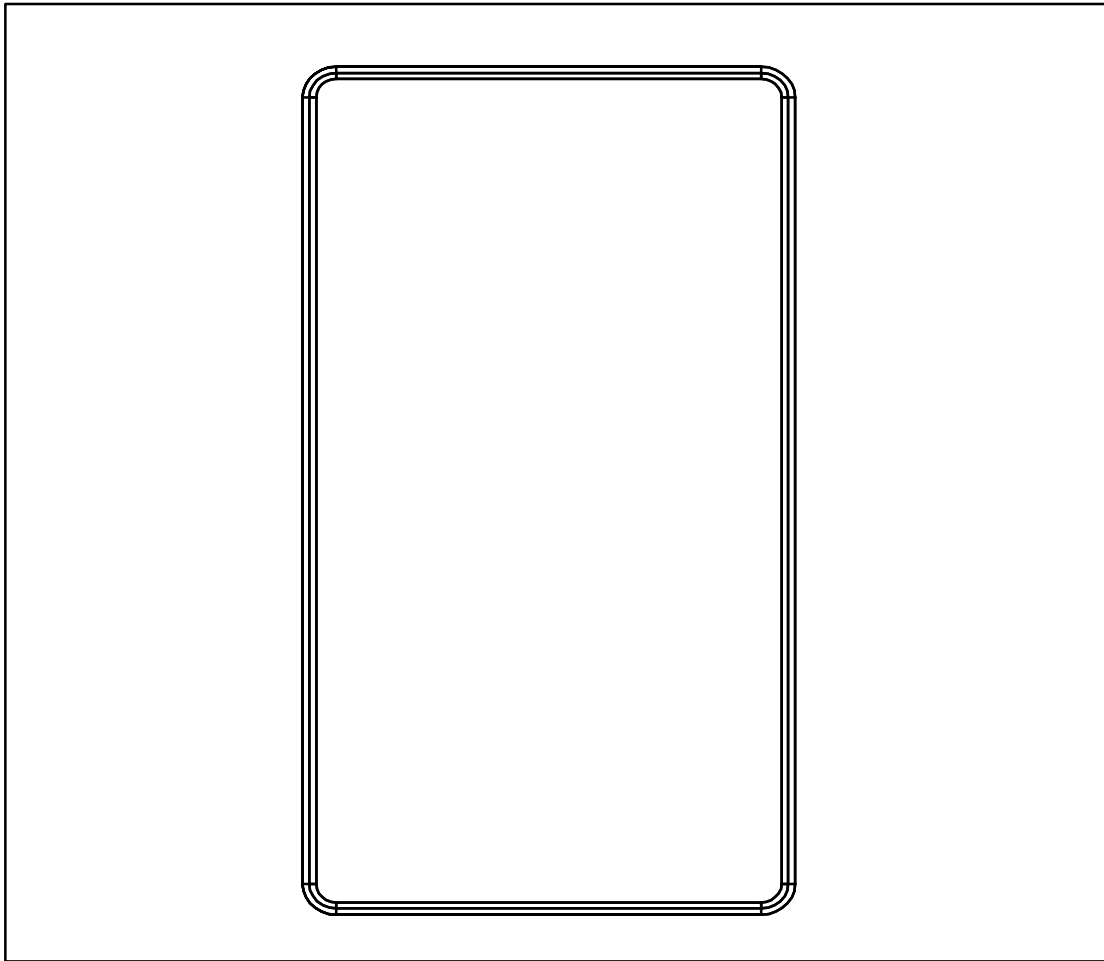


Figure 5–2 Calculator Top before pockets

Activity: Create the Calculator Top

Step 1 Copy the template file *calc_model_top.c* to your sub-directory.

Step 2 Edit the file and add the Open API function calls to create the block, blend the correct edges, and perform the hollow.

NOTE Do not uncomment the call to *calc_model_top_holes* at this time.

Step 3 Edit your *calc_model.c* routine and uncomment the call to *calc_model_top*.

Step 4 Edit *calcproto.h* and add the prototype for *calc_model_top*.

Step 5 Edit the *Makefile* and add *calc_model_top.o* to the SUBOBJS list and make a new executable.

Do NOT save the part files created!

Code Discussion: calc_model_top_holes.c

```

/* Get the body faces. Obtain the count of faces (count) */
flag = UF_CALL( UF_MODL_ask_body_faces(body,&list1) );
if(flag) return (flag);

flag = UF_CALL( UF_MODL_ask_list_count(list1,&count) );
if(flag) {
    UF_CALL( UF_MODL_delete_list( &list1 ) );
    return (flag);
}

/* Loop through the faces and check the face_data for a planar face
 * (UF_bounded_plane_subtype) with a normal in the +Z
 */
for(i=0; i < count; i++) {
    tag_t
    face;

    int ftype,          /* face type */
        dsense;       /* face direction sense */

    double
        pt1[3],       /* coordinates of vertex */
        nrm[3],       /* Face normal direction */
        box[6],       /* face boundary */
        rad1,         /* face major radius */
        rad2;         /* face minor radius */

    /* Extract a face from the list. Check the return code. */
    flag = UF_CALL( UF_MODL_ask_list_item(list1,i,&face) );
    if(flag) {
        UF_CALL( UF_MODL_delete_list( &list1 ) );
        return (flag);
    }

    /* Obtain the face data. Check return code. */
    flag = UF_CALL( UF_MODL_ask_face_data(face,&ftype,pt1,nrm,box,
        &rad1,&rad2,&dsense) );
    if(flag) {
        UF_CALL( UF_MODL_delete_list( &list1 ) );
        return (flag);
    }
    if(ftype == UF_bounded_plane_subtype) {
        if(fabs(nrm[2] - 1.0) < 0.001) {
            facel = face;
            break;
        }
    }
}
}

```

The above code finds the face upon which the pocket features will be placed. A list of faces is created. Then the list is cycled until a face is found that is a bounded plane (`if(ftype == UF_bounded_plane_subtype)`) and has a normal of $Z=1.0$.

```

/* Evaluate the expressions for Body_width, Body_height, pc_height,
 * lcd_height, lcd_pc_height, Button_width, and Button_height. Check
 * return codes
 */
    flag = UF_CALL( UF_MODL_eval_exp("Body_width", &body_width ) );
    .
    .
    .

```

In order to create our pocket features, calculations must be made based upon our expressions. *UF_MODL_eval_exp* takes the name of an expression and returns its numerical value. We need the values of **seven** of our expressions.

```

/* Create the 1st button pocket. Use a corner and floor radius of
 * "0.0" and a taper angle of 1.5. Check the return code.
 */

    strcpy(po clen[0], "Button_width+0.5");
    strcpy(po clen[1], "Button_height+0.5");

    location[0] = 5.0 + 0.5*(button_width);
    location[1] = 5.0 + 0.5*(button_height);

```

A pocket is initially placed by its center. Here, we set the location (using the data from the evaluated expressions) so that the left and bottom edges are five millimeters from the origin of the working coordinate system. We will want to know where these edges are so that we can find them later.

```

flag = UF_CALL( UF_MODL_create_rect_pocket(location,axis,x_dir,
    po clen_p, "0.0", "0.0", "1.5", face1, &feat1) );

```

Here is where the first of three pockets, the photocell pocket, is created. The feature location is the center of the top of the pocket. The axis points directly into the placement face. The X, Y, and Z lengths are set by expressions. The Z length is set to 2 so that the pocket will go all the way through the top half body.

The side and floor radii are expressions set to zero and the taper is an expression set to 1.5 degrees. The taper angle is set because this part will be made by injection molding and the taper is needed for the part to be ejected from the mold.

For the pocket feature, tapers go inward. So this pocket will be larger on the placement face than on the back face. Negative tapers are **not** permitted with the rectangular pocket creation function.



```
flag = UF_CALL( UF_MODL_create_linear_iset(0, "Across", bspacex,  
                                             "Up", bspacey, list1, &feat1) );
```

Create a linear instance set (rectangular feature instance array) of the button hole using the general method. The expressions **Across** and **Up** control the number of holes created. **bspacex** and **bspacey** are defined in the previous lines of code and are used so that this command is more readable. The first button hole feature has been placed in the list **list1**.

The *calc_model_top_holes* program will result in a top_half component that appears as follows:

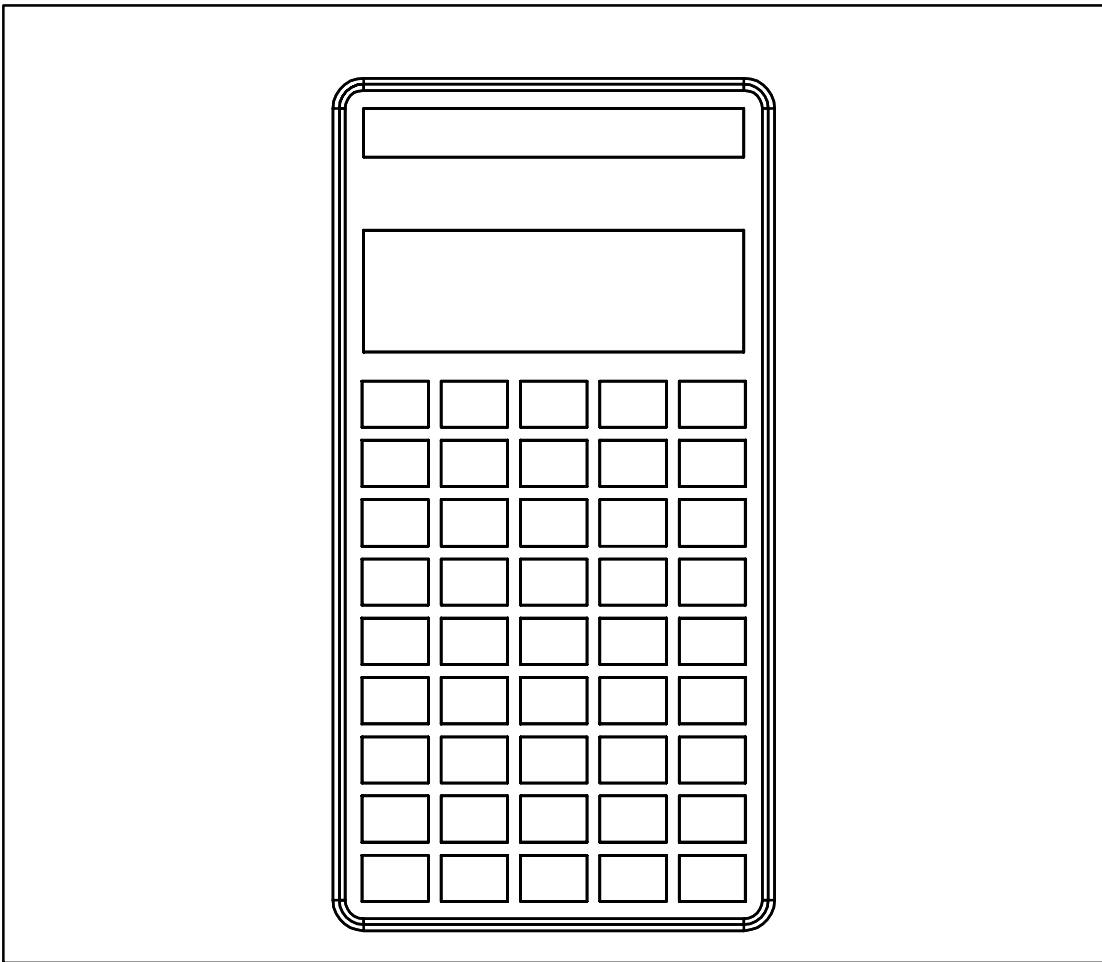


Figure 5–3 Calculator top with pockets



Activity: Add Holes to the Calculator Top

- Step 1** Copy the template file *calc_model_top_holes.c* to your sub-directory.
- Step 2** Edit the file and add the Open API calls to create the pockets (holes).
- Step 3** Edit your *calc_model_top.c* routine and uncomment the call to *calc_model_top_holes*.
- Step 4** Edit *calcproto.h* and add the prototype for *calc_model_top_holes*.
- Step 5** Edit the *Makefile* and add *calc_model_top_holes.o* to the SUBOBJS list and make a new executable.

Do NOT save the part files created!

Calculator Bottom Half

The bottom half of the calculator starts with a primitive block. It is blended and hollowed similarly to the top half. No pockets (photocell or button holes) are created in the bottom half component.

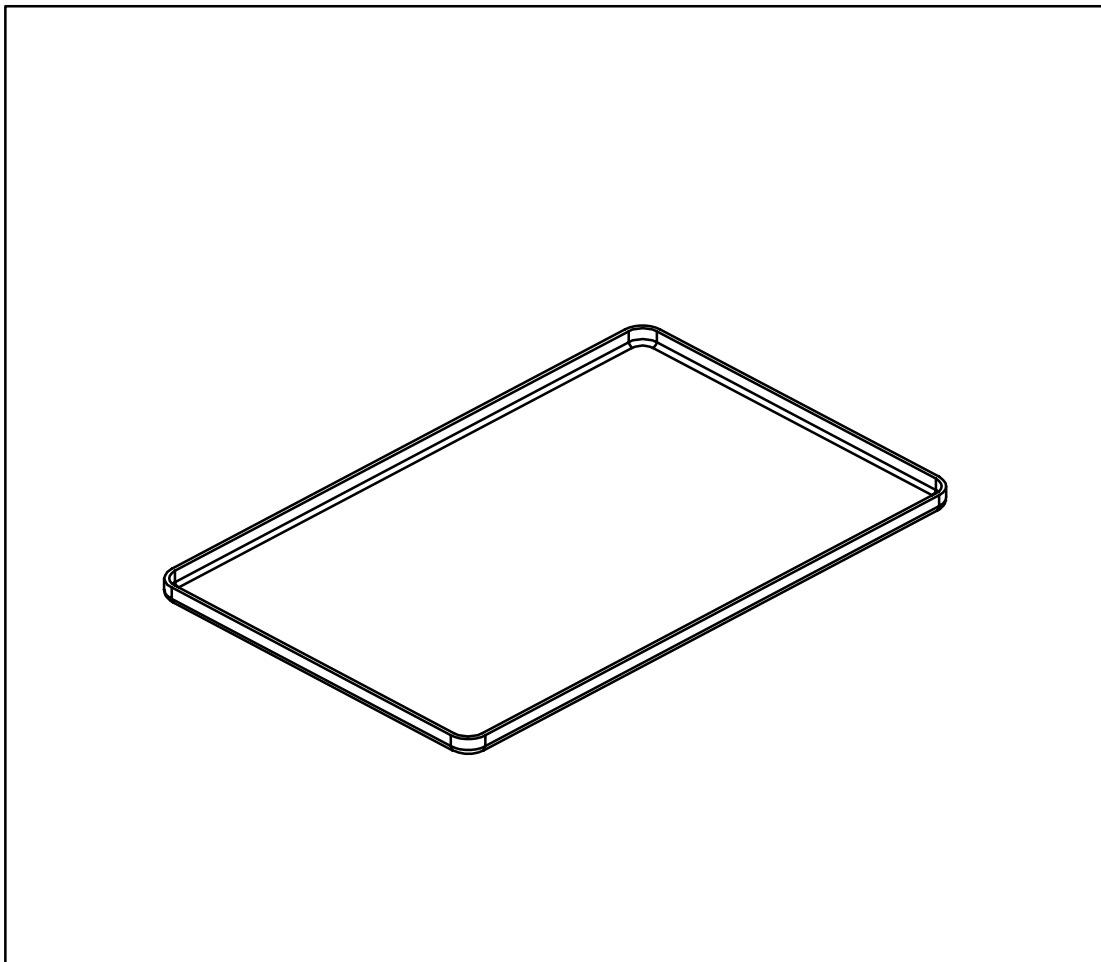


Figure 5–4 Calculator bottom half

Code Discussion: calc_model_bottom.c

The calculator bottom routine does not introduce any different concepts from those discussed in the calc_model_top.c section. See the discussion on calc_model_top.c if you have any questions.

Calculator Button

The button will be a simple block with a through slot cut in it. The through slot can be used to separate the standard operation of a button from its shifted or “2nd” operation.

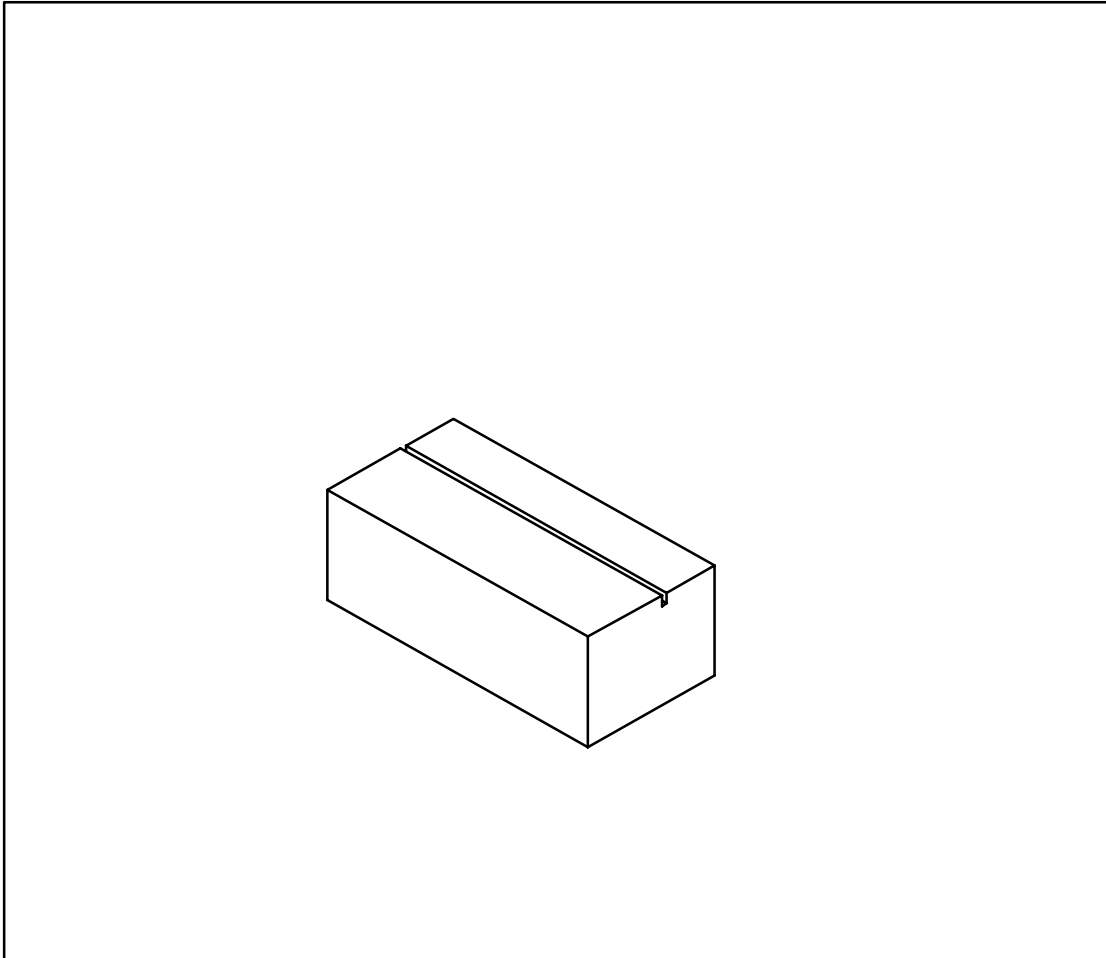


Figure 5–5 Button component

Code Discussion: calc_model_button.c

This function has similarities to the previous routines that created blocks and added features. It searches the body to find the slot limits by finding through faces (face1 and face2) by examining normals. The placement face (face3) is also determined by its normal vector orientation.

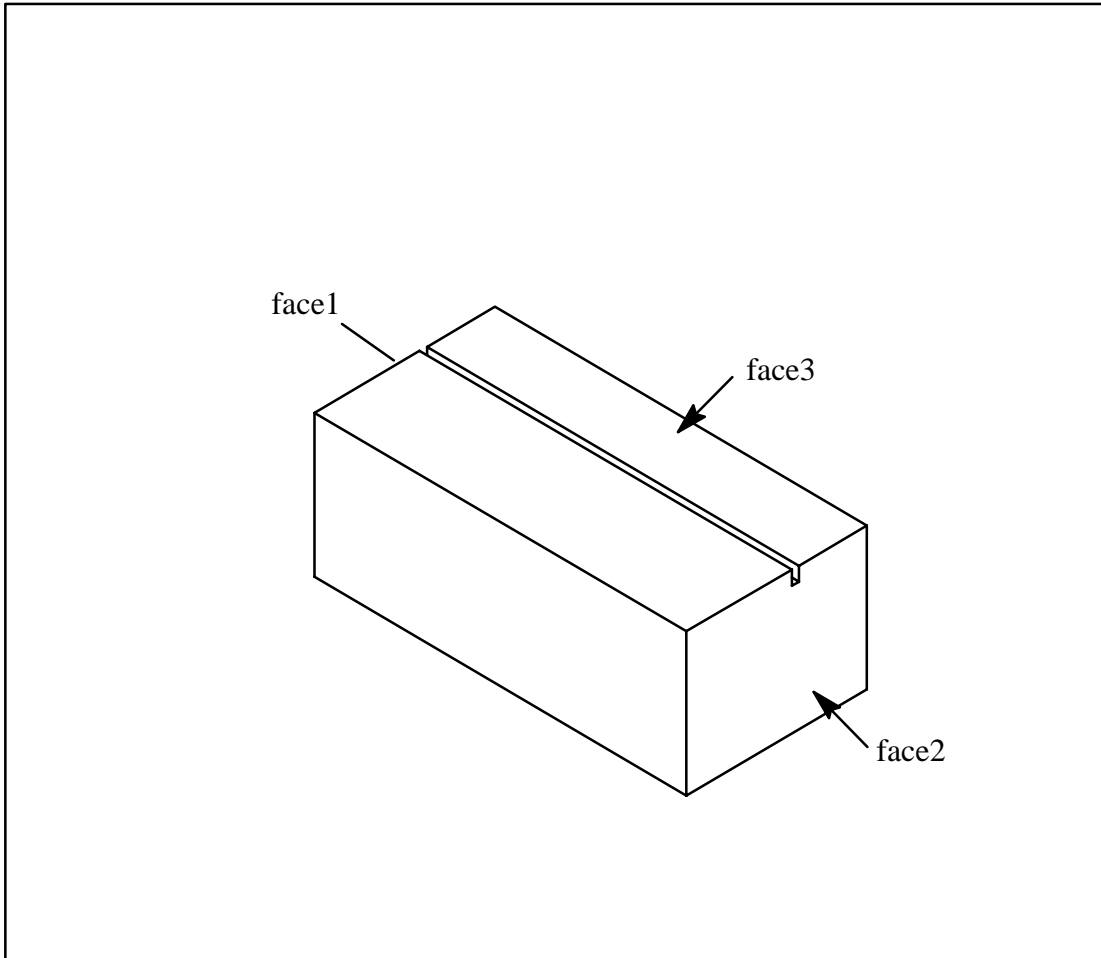


Figure 5–6 Faces used in slot feature creation

```
axis[0] = 0.0;  
axis[1] = 0.0;  
axis[2] = 1.0;
```

```
x_dir[0] = 1.0;  
x_dir[1] = 0.0;  
x_dir[2] = 0.0;
```

In this case, the tool axis points in the same direction as the normal to the placement face. The slot direction is specifies as a vector in the X direction.

Activity: Create Bottom Half and Button Geometry

- Step 1** Copy the template file *calc_model_button.c* to your sub-directory.
- Step 2** Edit the file and add the Open API function calls needed to create the geometry.
- Step 3** Copy the completed file *calc_model_bottom.c* from the solution directory.
- Step 4** Uncomment the calls to the button routine bottom routines in *calc_model.c*
- Step 5** Edit *calcproto.h* and add the prototypes for the button and bottom routines.
- Step 6** Edit the *Makefile* and add *calc_model_button.o* (and *calc_model_bottom.o*, if necessary) to the SUBOBS list and make a new executable.

Do NOT save the part files created!

Code Discussion: calc_comp_array.c

```

linear_mc.array_subtype = UF_ASSEM_linear_array;
linear_mc.master_component = NULL_TAG;
linear_mc.template_component = button_inst;
strcpy( inst_array_name, "BUTTON_ARRAY");
linear_mc.array_name = inst_array_name;

```

We will be creating a linear array of the button instance and naming it “BUTTON_ARRAY”. The `master_component` member of the structure is not used for this type of component array.

```

/* Set the linear_mc.dimensions[0..1] values to the tags of the
 * Across and Up expressions.
 */
flag = UF_CALL(UF_MODL_create_exp_tag( "Across",
                                     &(linear_mc.dimensions[0]) ));
if(flag) return (flag);
flag = UF_CALL(UF_MODL_create_exp_tag( "Up",
                                     &(linear_mc.dimensions[1]) ));
if(flag) return (flag);

```

The structure refers to the expressions using a tag rather than a character string as seen in the expression lesson. Here we create the tags from the names.

```

/* Create the master component array. Check return codes. */
flag = UF_CALL(UF_ASSEM_create_mc_array( &linear_mc, &array ) );
if(flag) return (flag);

```

This creates an array of components using the same expressions that were used to create the button hole feature `iset`. Another method, associating a component with an instance array, was not used, because that method requires the component be mated to a face that belongs to the feature instance array only.

Activity: Create Routine `calc_comp_array`

Step 1 Copy the template file `calc_comp_array.c` to your sub-directory. Edit the program and add the appropriate Open API calls.

Step 2 Uncomment the call to `calc_comp_array` in your `calc_model` program.

Step 3 Edit `calcproto.h` and add the prototype for the new routine.

```
int calc_comp_array( tag_t );
```

Step 4 Edit the *Makefile* and add *calc_comp_array.o* to the **SUBOBJS** list and make a new executable. Use the **Assembly Navigation Tool (ANT)** to view the assembly part structure created.

Do not save the part files created!

A hidden edge removed TFR–ISO view of the model should display something similar to the following.

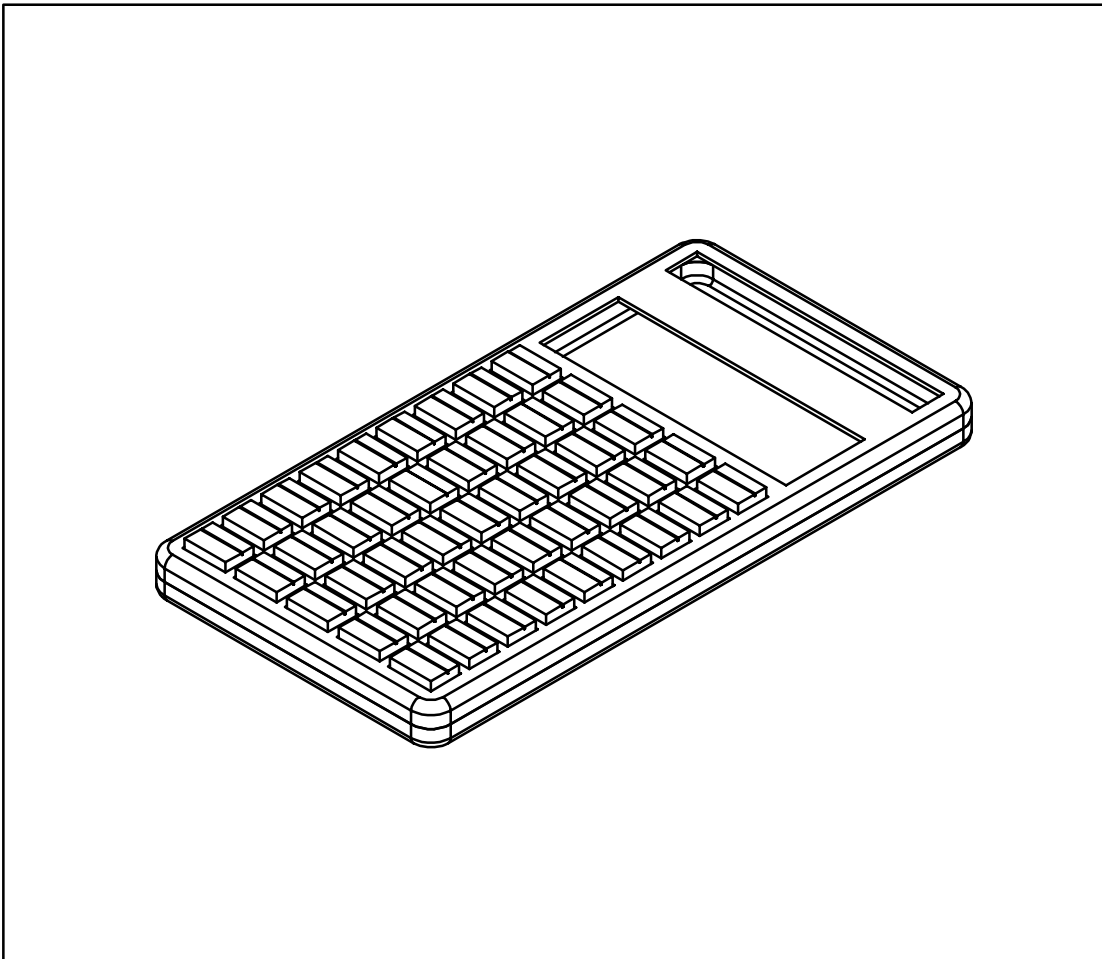


Figure 5–7 Completed Calculator model

SUMMARY

You created a parametric solid bodies using feature based modeling.

In this lesson, you :

- Exported expressions from the master part and imported to component parts.
- Created parametric solid primitives.
- Applied form features to your solids.
- Cycled the data model for bodies
- Cycled through faces and edges to identify desired geometry.

(This Page Intentionally Left Blank)



Dimensioning

Lesson 6



Drafting dimensions can be created on the drawing or on a model view before the view is imported to a drawing. This lesson discusses Dimension and Drafting Aid creation, starting with the Open API routines to read and set the Drafting application preferences. The description for the drafting parameter arrays is not included in this text. It is available from the Open API Reference documentation in the Drafting section.

OBJECTIVES

Upon completion of this lesson, you will be able to:

- Read and set the drafting preferences.
- Create linear dimensions on a model or drawing view.
- Create notes and labels
- Control the WCS and create new coordinate systems.
- Create, retrieve, and save views and layouts.

Dimension and Drafting routines

6

The following are a sample of the functions defined in `uf_drf.h`:

- `UF_DRF_ask_preferences`
- `UF_DRF_set_preferences`
- `UF_DRF_create_horizontal_dim`
- `UF_DRF_init_object_structure`
- `UF_DRF_create_vertical_dim`
- `UF_DRF_create_note`
- `UF_DRF_create_label`
- `UF_VIEW_ask_tag_of_view_name`
- `UF_VIEW_expand_view`
- `UF_VIEW_is_expanded`
- `UF_VIEW_unexpand_work_view`

calc_dimension.c

This routine controls the creation of dimensions for the objects from the three components. I simply calls a dimensioning routine for each of the three components.

Most of the dimensions will be placed on the top half of the calculator. The edges necessary for dimensioning are obtained in another routine.

Dimension locations are specified in the template files for dimension creation.



Code Discussion: calc_dimension_top.c

```
/* Read the current settings for drafting/dimensioning */
   flag = UF_CALL( UF_DRF_ask_preferences (mpi,mpr,radsym,diasym) );
```

UF_DRF_ask_preferences reads all of the creation parameters for dimensions and drafting aids and loads them into local variables. **mpi** is an integer array dimensioned to 100, **mpr** is a double precision array dimensioned to 70, and **radsym** and **diasym** are arrays of six characters each that hold the radius and diameter symbol strings.

```
/* Set the linear units to mm and change the character size and number
 * of decimal places.
 * Call the routine that reset draft/dim settings.
 */
   mpi[3]   = 2;           /* decimal places */
   mpi[13]  = 1;           /* linear units */
   mpr[32]  = 5.0;        /* character height */
   flag = UF_CALL( UF_DRF_set_preferences (mpi,mpr,radsym,diasym) );
```

UF_DRF_set_preferences writes drafting creation parameters back to the data model. This body of code changes decimal places to two, linear units to millimeters, and character height to five millimeters for all dimensions and drafting entities that are going to be created.

```
/* Initialize structures. This sets the associativity modifiers
 * to UF_DRF_end_point (which is what we want. Check return codes!
 * Also initialize the text structure.
 */
   flag = UF_CALL( UF_DRF_init_object_structure( &dim0 ) );
   if( flag ) return( 1 );
   flag = UF_CALL( UF_DRF_init_object_structure( &dim1 ) );
   if( flag ) return( 1 );
   dim_text.lines_app_text = 0;
   dim_text.user_dim_text = NULL;
   dim_text.appended_text = NULL;
```

The structures used for the object information for the dimension are initialized. An Open API function call is used for the object structure. The text structure is initialized explicitly.

```
/* Get the edges for the dimensions. They are returned in a particular
 * order documented in the comments at the routine start.
 */
   flag = calc_dimension_top_edges (edge, epcode);
   if(flag) return (flag);
```

calc_dimension_top_edges returns a list of edge objects and a list of endpoint codes (either first point or last point) to be used for the dimensions on the edges. The code will be examined later in this lesson.

```

/* Obtain the tag of the work view.  It is used in the structures. */
flag = UF_CALL( UF_VIEW_ask_tag_of_view_name( "", &view_tag ) );
if(flag) return (flag);
dim0.object_view_tag = dim1.object_view_tag = view_tag;

```

The dimension structure requires the tag of the view containing the object. This is necessary when dimensioning on a drawing. The tag of the object being dimensioned exists in model space. When dimensioning on a drawing, the tag of the member view is needed so Unigraphics can map from model space to drawing space. We are applying our dimensions in the model view. The tag of the work view is loaded in the structure to indicate that no mapping is necessary.

```

/* First dim. is from left edge to right.  Set the location. */
origin[0] = 0.5 * width;
origin[1] = height + 6.0*mpr[32];
origin[2] = 5.0;

```

The location of the dimension is determined from the calculator width and dimension text size. **mpr[32]** is the dimension text size. The origin assignments set this dimension midway between the left and right edges of the part (X), at six times the character height above the part (Y), and on the bottom face of the top half of the calculator (Z).

```

/* Set the structure data for both object tags and assoc. modifiers.
 * Create the dimension.
 */
dim0.object_assoc_modifier = ecode[0];
dim0.object_tag = edge[0];
dim1.object_assoc_modifier = ecode[1];
dim1.object_tag = edge[1];
flag = UF_CALL( UF_DRF_create_horizontal_dim( &dim0, &dim1,
&dim_text, origin, &dimension ) );

```

This body of code creates the first of our dimensions. The dimension structure objects are set. The modifiers that indicate which end to use for the extension line are also set.



Code Discussion: calc_dimension_top_edges.c

6 *calc_dimension_top_edges* finds which edges are to be dimensioned and returns them sequentially with a flag to indicate which end of the edge to use. The completed function is provided.

```
/* Evaluate expressions for calculator */
   flag = UF_CALL( UF_MODL_eval_exp("Body_height",&height) );
   if(flag) return(flag);
```

The edges to be dimensioned will be found by their properties of length and location. The lengths and locations are found by evaluating the expressions.

```
/* Locate the target body using it's name; TOP_HALF */
   body = NULL_TAG;
   flag = UF_CALL( UF_OBJ_cycle_by_name("TOP_HALF",&body) );
```

UF_OBJ_cycle_by_name, when used as above, finds the first object with the specified name. The function is normally called in a loop but we know there is one body called "TOP_HALF" in the part.

TIP It's a good idea to name the solids you create. This is easy to do when a program is doing the work of applying the system \$NAME attribute. You never know when the names will come in handy. They may be used for finding solids or later in creating reports.

```
/* Obtain the edges and count of edges in the body. */
   flag = UF_CALL( UF_MODL_ask_body_edges(body,&elist) );
   if(flag) return(flag);

   flag = UF_CALL( UF_MODL_ask_list_count(elist,&ecount) );
   if(flag) return(flag);

/* Loop through the edges and find the vertices. Check for specific
 * edges and store those particular edges and the appropriate end
 * point code. UF_DRF_first_end_point if the first vertex is the
 * desired endpoint; UF_DRF_last_end_point for the second vertex.
 */
   for(i = 0; i < ecount; i++) {
       flag = UF_CALL( UF_MODL_ask_list_item(elist,i,&edge_tag) );
       if(flag) return(flag);

       flag = UF_CALL( UF_MODL_ask_edge_verts(edge_tag,pt1,pt2,&vcount)
   );
       if(flag) return(flag);
```

```

/* Check for vertical edges. */
    if((fabs(pt1[0] - pt2[0]) < 0.001)) {

/* Look for left vertical edge at the back */
    if((fabs(pt1[0]) < 0.001) && (fabs(pt1[2] - 5.0) < 0.001)) {
        edge[0] = edge_tag;
        if(pt1[1] > pt2[1])
            ecode[0] = UF_DRF_first_end_point;
        else
            ecode[0] = UF_DRF_last_end_point;
        continue;
    }
}

```

A list of edges is obtained and the routine cycles the list looking for edges that match certain qualifications. When an edge is found, it is added to the edge array returned through the argument list. The startpoint/endpoint code is set for the edge. It, too, is returned through the argument list.

All the vertical edges are stored in the list first, followed by the horizontal edges. As always, when checking two floating point numbers for equivalence, do not check them against one another but check the absolute value of their difference against some tolerance.

As Open API application programmers, you will have to understand assemblies at a deeper level than an external user. Here is an example where the difference between prototypes and occurrences is critical.

```

/* The edges procured are the prototype (objects in the child). Find
 * the occurrence tags of these objects so the dimensions will be made
 * to objects in the assembly.
 */
    for(i=0; i < 14; i++) {
        UF_ASSEM_ask_occs_of_entity(edge[i], &occs);
        edge[i] = occs[0];
        UF_free(occs);
    }
}

```

When a component is brought into an assembly, the geometry in the child part is *prototype* geometry. The geometry in the component (in the assembly) are object occurrences.



When the routine *UF_OBJ_cycle_by_name* was used

```
flag = UF_CALL( UF_OBJ_cycle_by_name("TOP_HALF", &body) );
```

to get the body of the top half, the work part was the assembly, so it returned the identifier of the object occurrence of the body in the assembly.

When the occurrence of the body was used to get the list of edges with

```
flag = UF_CALL( UF_MODL_ask_body_edges(body, &elist) );
```

Unigraphics returned the identifiers of the prototypes of the edges.

To get the occurrence of the object, we used

```
UF_ASSEM_ask_occs_of_entity(edge[i], &occs);
```

for each prototype edge to get the associated object occurrence in the assembly. Remember, the dimensions are being created in the assembly.

calc_dimension_bottom.c

There are no dimensions for the bottom half of the calculator. The *calc_dimension_bottom* function currently returns with no actions.



Activity: Create Top Dimensions

6

- Step 1** Copy the files *calc_dimension*, *calc_dimension_top*, *calc_dimension_bottom*, and *calc_dimension_top_edges* to your sub-directory.
- Step 2** Add the necessary Open API function calls to *calc_dimension_top*.
- Step 3** Add the prototypes to *calcproto.h*.
- Step 4** Change the *calculator* routine to call the *calc_dimension* routine.
- Step 5** Edit *Makefile* and add *calc_dimension.o*, *calc_dimension_top.o*, *calc_dimension_top_edges.o*, and *calc_dimension_bottom.o* to the SUBOBJS list and make a new executable. You should see dimensions on the calculator in the TOP view.

Remember, do NOT save your part files!

The previous activity will generate the dimensions in the TOP view of the assembly. It will look similar to the following.

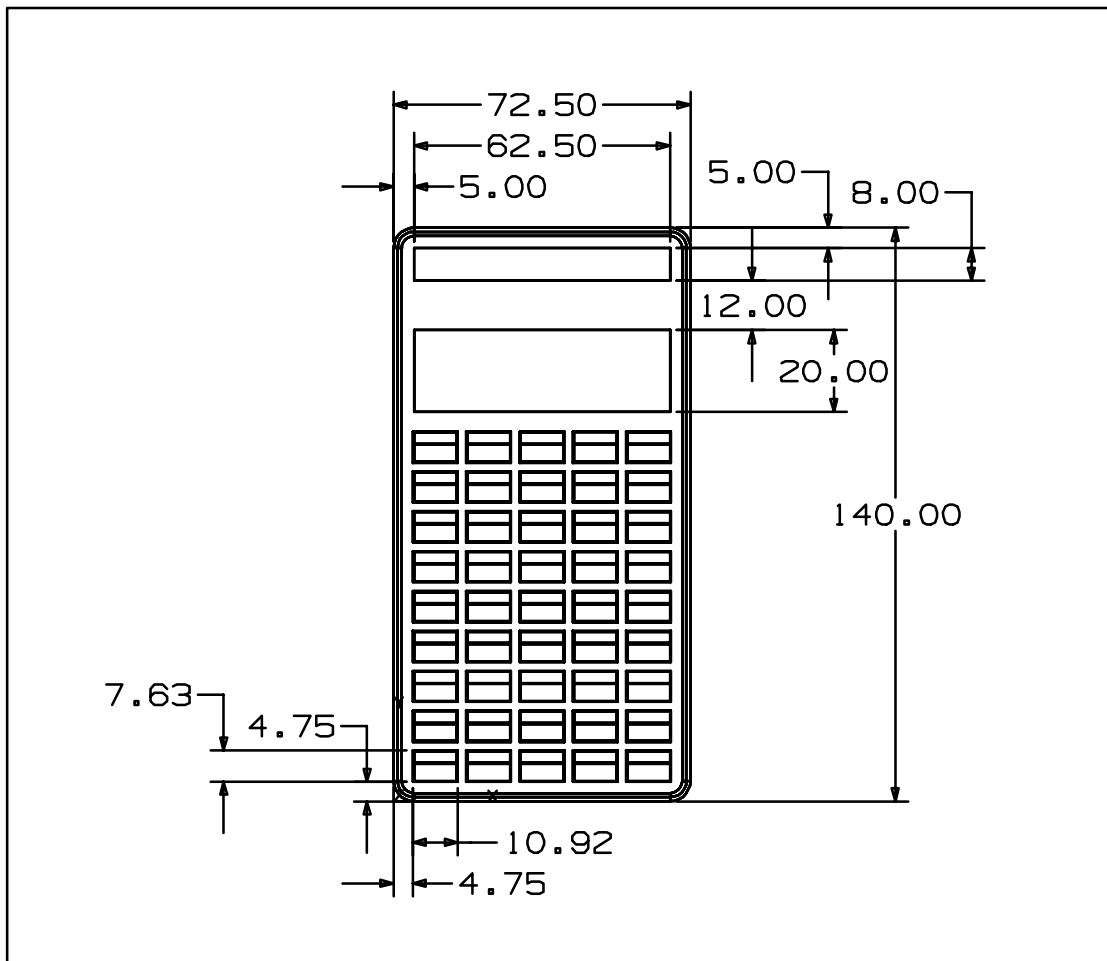


Figure 6-1 Calculator Top dimensions

These are the default dimensions. If you vary the dialog input, your dimensions will be different.

WCS control routines

The following routines allow control of the WCS. Prototypes are in `uf_csys.h`.

- **UF_CSYS_create_matrix**
- **UF_CSYS_create_csys**
- **UF_CSYS_create_temp_csys**
- **UF_CSYS_set_wcs**
- **UF_CSYS_ask_wcs**
- **UF_CSYS_map_point**

View/Layout control routines

The following routines manipulate views and layouts. Prototypes are in `uf_layout.h` or `uf_view.h`.

- **uc6464** **Replace View in Layout**
- **uc6449** **Change Work View**
- **uc6466** **Read Current Layout Name and Work View Name**
- **uc6468** **Retrieve Layout**
- **uc6430** **Read View Center and Scale**
- **uc6431** **Set View Center and Scale**
- **uc6432** **Fit the View(s)**
- **UF_VIEW_fit_view**
- **uc6433** **Read View Matrix**
- **uc6434** **Set View Matrix**
- **uc6442** **Read View Drawing Parameters**
- **uc6443** **Set View Drawing Parameters**
- **UF_VIEW_ask_work_view**
- **UF_VIEW_save_all_active_views**
- **uc6450** **Save View**
- **uc6454** **Delete View**
- **UF_VIEW_delete**
- **UF_VIEW_cycle_objects**

Code Discussion: calc_dimension_button.c

6

The routine *ug_set_wcs_to_view* changes the WCS pointer (makes the WCS point to a different csys object). The csys objects created to allow dimension creation in the RIGHT and FRONT views should be deleted at the end of the routine.

```
/* Set the work view (and WCS) to create a dimension in the FRONT
 * view. Save the original WCS.
 */
UF_CALL( UF_CSYS_ask_wcs(&csys1) );
uc6464( "", "", "FRONT" );
flag = ug_set_wcs_to_view("FRONT");
if(flag) return(flag);
```

The point of this body of code is to set the view and the Work Coordinate System (WCS) for the dimensions that are going to be created. The dimensions are placed in the XC–YC plane or work plane of the WCS. The WCS must be oriented to the view coordinate system to have the dimensions created in the proper orientation to the view.

UF_CSYS_ask_wcs returns the tag to the current WCS. This is saved so the WCS can be restored when the dimensioning is done. The intermediate WCS display artifacts created are temporary and will disappear when the screen is refreshed.

uc6464 replaces a view in a layout. The null strings for the first two arguments indicate that the current layout work view is to be replaced. The third argument is the name of the view to put into the layout.

ug_set_wcs_to_view.c

ug_set_wcs_to_view sets the WCS to the view passed through the argument list. The routine changes the WCS pointer to a coordinate system (csys) created from the view matrix.

6

calc_dimension_button_edges.c

The completed routine to obtain edges for dimensioning the button is provided. The routine uses Open API function calls that have been discussed. It locates edges of the button and returns them in a particular order.

Activity: Create Button Dimensions

6

- Step 1 Copy the routines *calc_dimension_button*, *calc_dimension_button_edges*, and *ug_set_wcs_to_view* to your sub-directory.
- Step 2 Add the appropriate Open API function calls to *calc_dimension_button* and *ug_set_wcs_to_view*.
- Step 3 Uncomment the call to *calc_dimension_button* in the *calc_dimension* routine.
- Step 4 Update *calcproto.h* and add prototypes for the three new routines.
- Step 5 Edit *Makefile* and add *calc_dimension_button.o*, *ug_set_wcs_to_view.o* and *calc_dimension_button_edges.o* to the SUBOBJS list and make a new executable.
- Step 6 Run the program.
- Step 7 Change views (or open a four view layout) to see the dimensions on the FRONT and RIGHT views.

NOTE

If you used the provided solutions, you will have to open **L4_DRF**. This example is for many companies who desire that annotations not be placed on modeling views. The solutions contain extra steps to create three drafting views: **TOP_DRF** in *calc_dimension_top*, **FRONT_DRF** and **RIGHT_DRF** in *calc_dimension_button*. Layout **L4_DRF** is created in *calc_dimension_button* by uc6460 and the **L1** layout is retrieved by uc6468.

Remember, do NOT save your part files!

SUMMARY

You created dimensions on an assembly in three different model views. You controlled the WCS and layout to perform these activities.

In this lesson, you :

- Changed preferences for drafting object creation.
- Created horizontal and vertical dimensions.
- Replaced views in a layout.
- Created temporary coordinate systems.
- Changed the WCS orientation.



(This Page Intentionally Left Blank)

Drawings

Lesson 7

The drawing lesson will introduce you to the creation and modification of drawings. The process of adding views to the drawing, changing drawing member view characteristics and updating drawings is also covered.

OBJECTIVES

Upon completion of this lesson, you will be able to:

- Create a metric or english standard size drawing.
- Import model views to the drawing.
- Retrieve (import) a part as a drawing border.
- Cycle the member views of the drawing.
- Update views on the drawing.
- Cycle various object classes on a drawing.



Member View and Drawing Name Conventions

After the model is complete, the next step is to create a drawing. The dimensions for the class project were added to the model views and will be imported when the views are put onto the drawing. A drawing border will also be included.

In the Open API, drawing member view names need an “@” character to denote it as a drawing member view. Specifying the view name this way is actually a short cut since the actual view name contains an integer following the “@” character. Optionally, you may enter the full name in order to assure uniqueness of member view names. Drawings can have more than one instance of a given view on a single drawing in Unigraphics.

An example of a drawing member view name that is derived from the model view, TOP, would be “TOP@3”. To specify the name of the model view, “TOP” should be given. To specify the name of the drawing member view, either “TOP@” or “TOP@3” can be given. The full names of all the member views for a given drawing can be retrieved using uf6499.

Additionally, drawings have an associated drawing work view whose name is the drawing name with “@0” appended to it.

Functions that create, retrieve, or require the presence of a drawing will only be allowed in the Drafting module when running interactive Open API programs.

NOTE In Unigraphics, when the drawing view is the work view, the layout which contains the drawing is called !DRAWING. This layout name is read only and cannot be used to set the current layout.

NOTE In routines that specify open and closed quotes = current drawing, view name, etc., there are no spaces between the quotes (“”).

Drawing routines

The following routines relate to creating and working on Drawings and Member Views.

- **uc6476** **Set Drawing State**
- **uc6477** **Retrieve Drawing State**
- **uc6478** **Create Drawing**
- **uc6479** **Read Drawing Size**
- **uc6480** **Set Drawing Size**
- **UF_DRAW_ask_drawing_info**
- **UF_DRAW_set_drawing_info**
- **UF_DRAW_import_view**
- **uc6481** **Add View to Drawing**
- **uc6482** **Remove View from Drawing**
- **uc6483** **Read View Reference Point on Drawing**
- **uc6484** **Set View Reference Point on Drawing**
- **uc6485** **Read View Borders on Current Drawing**
- **uc6486** **Set View Borders on Current Drawing**
- **UF_DRAW_define_view_manual_rect**
- **UF_DRAW_update_one_view**
- **uc6492** **Read Current Drawing Name**
- **UF_DRAW_ask_current_drawing**
- **uc6496** **Rename Drawing**
- **uc6495** **Delete Drawing**
- **uc6499** **Cycle Views in Drawing**



Importing Parts

UF_PART_import

Drawing borders can be created in a part file drawing, imported from a drawing border template file, or called in as a pattern. The part import method will pull in a copy of the part file onto the drawing (or into the model, depending on the mode selected). Updates to the drawing border file will not be reflected in files that have already imported the border part. Drawing borders must be imported when the drawing is the work view and the view import mode (in the `part_modes` structure) is set to not import views.

The pattern method allows you to have a single source for part file drawing borders. Changes to that part's pattern data will be reflected in drawings using that pattern as a border. The pattern related Open API functions will not be discussed here.

Code Discussion: `calc_drawing.c`

```
/* Set up identity matrix for part import. */
amat[0] = 1.0; amat[1] = 0.0; amat[2] = 0.0;
amat[3] = 0.0; amat[4] = 1.0; amat[5] = 0.0;
aorg[0] = 0.0; aorg[1] = 0.0; aorg[2] = 0.0;
```

The import routine uses a matrix to position the object(s) being imported. This sets up an identity matrix (no rotation) and an origin of 0,0,0 (no translation).

```
/* Get the name of the current layout. */
uc6466(curlay, curview);
```

`uc6466` gets the name of the current layout and current work view. In case something goes wrong, we'll want to restore the layout.

```
/* Create a drawing using the input argument name. Check return
 * code. Restore the old layout and return if an error occurs.
 */
flag = uc6478(dname, two, dsize, htwd);
if(flag != 0) {
    uc6468(curlay, one, rdum);
    return (flag);
}
```

`uc6478` creates a drawing and replaces the current layout with it. As the second argument is “**two**”, the drawing will be in millimeters. Because the drawing size is one of the standard sizes (A2), the fourth argument (**htwd**) is not used.

```

/* Initialize the structure for part import. */
    pmod.layer_mode = IP_WORK;
    pmod.group_mode = IP_NOGROUP;
    pmod.plist_mode = IP_NONE;
    pmod.view_mode = IP_NO_VIEW;
    pmod.cam_mode = FALSE;

```

The import part characteristics are set here. The structure members are set to:

- Import data onto the work layer
- Do not import the object(s) as a group
- Do not import part list data
- Do not import views

The option for views is important when importing data onto a drawing. If this option was set to `IP_VIEW`, the data from the part file would be imported into the model. The `IP_NO_VIEW` setting will cause view dependent data to be created if the current work view is a drawing.

```

/* Retrieve (import) title block part. Check the return code.
 * Restore the old layout and return if an error occurs.
 */
    strcpy(pfile, "./dwg-a2border.prt");
    flag = UF_CALL( UF_PART_import( pfile, &pmod, amat, aorg, ascale,
                                   &grp ) );
    if( flag != 0 ) {
        uc6468(curlay, one, rdum);
        return (flag);
    }

```

The part file that is imported **may** require a full or relative path name if the file is not in the directory where UG was executed. If the import fails, the old model view will be restored. The drawing will still exist: it will be empty.

```

/* Place the views onto the drawing. */
    flag = calc_drawing_views(dname, dsize);
    return (flag);
}

```

The views will be added to the drawing in locations based on the drawing size. The `calc_drawing_views` routine will place the TOP, FRONT, RIGHT and TFR–ISO view on the drawing.



```
/* Obtain the drawing tag. Could also use UF_DRAW_ask_current_drawing */
curview[0] = '\\0';
drawing_tag = NULL_TAG;
UF_OBJ_cycle_by_name( dname, &drawing_tag );

/* Cycle through the views and update each view. */
while( TRUE ) {
    tag_t view_tag;

/* Get the member view from the current drawing. */
    uc6499( "", curview );
    if( '\\0' == curview[0] )
        break;
/* Get the tag of that view and update the view on the drawing */
    UF_CALL(UF_VIEW_ask_tag_of_view_name( curview, &view_tag ) );
    UF_CALL(UF_DRAW_update_one_view( drawing_tag, view_tag) );
};
```

The views will not be up-to-date on the drawing. We must cycle through all member views and update the views. The view and drawing tags are required. The *UF_OBJ_cycle_by_name* routine obtains the tag of the drawing. The *UF_VIEW_ask_tag_of_view_name* will return the member view tags. Views currently must be updated one at a time.

Code Discussion: calc_drawing_views.c

```
/* Evaluate the expressions for calculator height and width. Check
 * return codes.
 */
    flag = UF_CALL( UF_MODL_eval_exp("Body_height",&height) );
    if(flag) return(flag);
```

We will need the calculator dimensions in order to set up the view centers and to place the views on the drawing.


```

/* Calculate spacing between views */
    hspace = (htwd[dsize-1][1] - width - height)/3.0;
    vspace = (htwd[dsize-1][0] - height - 12.0)/3.0;

```

These calculations are designed so that the space between the views is equal to the space between a view and the nearest border. Note that the drawing height (Y) is the first position in the array and width (X) is the second position. The horizontal space is one-third of the entire drawing width minus the calculator width (which is the horizontal display in the FRONT view) minus the calculator height (which is the horizontal in the RIGHT view). The vertical space is one-third of the drawing width minus the calculator height (vertical in the TOP view) minus 12.0 (vertical in the FRONT view).

The “12.0” is the depth of the calculator. There are ten millimeters for the calculator body and the buttons extend two millimeters beyond the body.

```

/* Set the View center and view's drawing ref. for the FRONT view. */
    center[0] = 0.5*width;
    center[1] = 0.5*height;
    center[2] = 6.0;
    flag = uc6431( "FRONT", center, 1.0 );
    if( flag ) return( flag );
    flag = uc6443("FRONT", center, 0.0 );
    if( flag ) return( flag );

```

The geometry was set up so that all measures and expressions dealt with positive values. In order for the views to line up correctly on the drawing, the view centers must be set to the center of the geometry. The view's reference point for placement on the drawing must also be set.

```

/* Import (add) the FRONT view to the drawing. Set refpt (the drawing
 * reference point) to position view on drawing. Check return code.
 */
    refpt[0] = hspace + width/2.0;
    refpt[1] = vspace + 6.0;
    flag = uc6481(dname, "FRONT", refpt, 1);
    if( flag ) return( flag );

```

Here is where the first view, the FRONT view is placed on the drawing. The refpt drawing location is where the view's drawing reference will be placed. Similar code will set the import location on the drawing (refpt) and add the view to the drawing (transferring view dependent geometry such as dimensions to the drawing/member view).



Activity: Create a Drawing

- Step 1 Copy the template files *calc_drawing_views.c* and *calc_drawing.c* to your sub- directory.
- Step 2 Edit the routines and add the appropriate Open API function calls.
- Step 3 Add a call to *calc_drawing* in the *calculator* program. You must strecpy a drawing name into the *dname* variable.
- Step 4 Add the prototype for *calc_drawing* and *calc_drawing_views* in the *calcproto.h* file.
- Step 5 Edit *Makefile* and add *calc_drawing_views.o* and *calc_drawing.o* to the SUBOBJS list and make a new executable. The drawing should appear as follows:

7

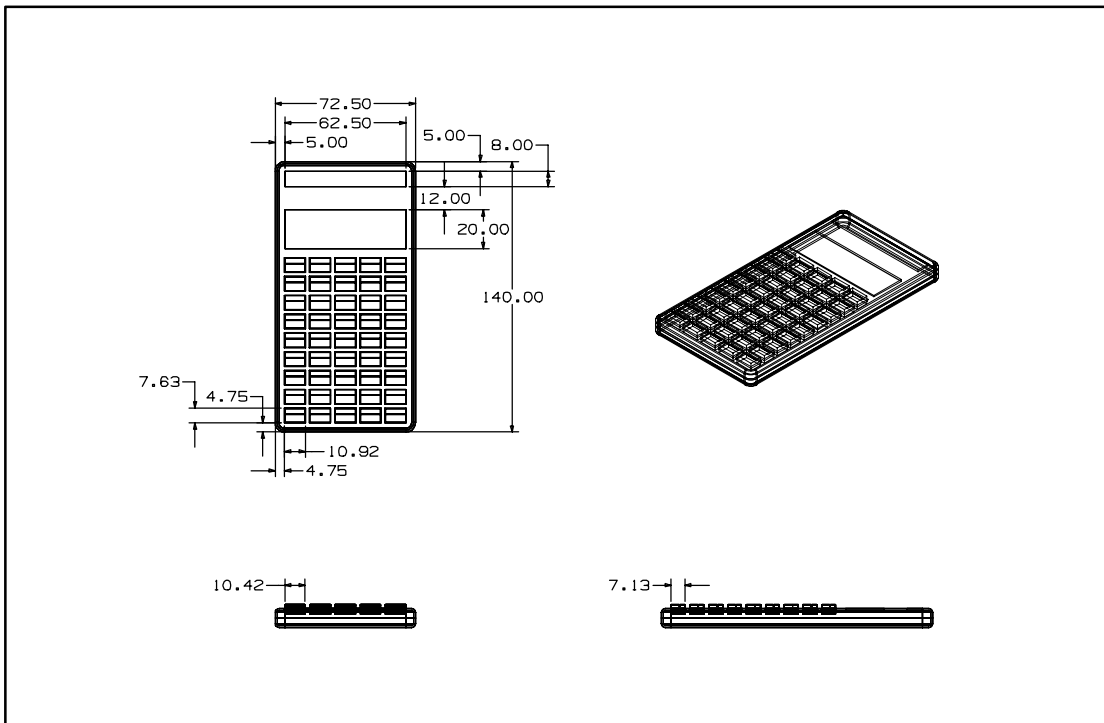


Figure 7-1 Calculator Drawing *(without border)

SUMMARY

You created a drawing using model views that contained dimensions. You also imported a drawing border.

In this lesson, you :

- Created a metric A2 sized drawing.
- Imported the model views at specific locations.
- Cycled the drawing member views and updated the out-of-date views.
- Imported drawing borders.





(This Page Intentionally Left Blank)

External Open API

Lesson 8

External Open API application programs allow you to execute a program without the graphics window display. There is little difference between internal and external programs. Part and object attributes are important for storing and retrieving information in the data model.

OBJECTIVES

Upon completion of this lesson, you will be able to:

- Create an external Open API application program.
- Generate attributes on parts and objects.
- Find and read attribute information.
- Identify some of the CFI (Common File Interface) functions.

External Open API Programs

An external Open API application program is a program that does not use Unigraphics to display a part file or dialogs. There is no user interaction unless it is done with printf/scanf or through a Motif dialog .

- The User Interface Styler is for internal programs only.
- A link flag (“-e” using ufmenu, “make -ext” using the makefile) is the only procedural difference in compiling and linking external programs.
- External programs start with `main(argc, argv)` rather than `ufusr(param, flag, plen)`.
- An external program ends with `exit()` rather than `return()`.

Some of the uses for an external program are:

- Checking parts to see if they follow standards and then either correcting or reporting problems.
- File management operations such as moving files from development to release directories.
- Reading attributes and writing reports.

Our external program opens the calculator part created by the programming in this course and writes a simple report based upon assigned attributes. In order to use this program, we will have to add attributes to the parts and save the parts created by the internal Open API application program. One way to do this is to select **File-Save All and Exit** after the calculator internal Open API application is run. This will allow us to finally have completed and saved part files at this point.

Attribute Routines

The following data structures are used by the UF_ATTR C routines. The full structure is defined in `uf_attr.h`. Field descriptions are found in the documentation under *Types*.

- UF_ATTR_part_attr_s
- UF_ATTR_value_s
- UF_ATTR_value_u

The following is a list of functions to set, modify, or access attributes:

- UF_ATTR_ask_part_attribute
- UF_ATTR_assign
- UF_ATTR_ask_part_attrs
- UF_ATTR_ask_part_attrs_in_file
- UF_ATTR_read_value
- UF_ATTR_cycle
- UF_ATTR_delete
- UF_ATTR_delete_all



Adding Attributes to the Calculator Parts

Attributes can be added to parts and objects at any time. Internal and external Open API programs can create, access and delete attributes. A section of code in our routines was omitted until the modeling functions were completed. These sections added attributes to the parts. The following code adds a part attribute to the top part.

```
char  svalue[50]
UF_ATTR_value_t att_value;  /* attribute data structure */
```

Declare a variable as the structure for attribute assignment.

Add part number and description attributes to the part. Use `UF_ATTR_ask_part_attribute` to obtain the part id for attribute assignment. Use `UF_ATTR_assign` to add the attribute. Use the titles PARTNUMBER and DESCRIPTN.

```
UF_CALL( UF_ATTR_ask_part_attribute( &atteid ) );
att_value.type = UF_ATTR_string;
strcpy(svalue,"001"); /* First call is the part number. */
att_value.value.string = svalue;
UF_CALL( UF_ATTR_assign( atteid, "PARTNUMBER", att_value ) );

strcpy(svalue,"TOP"); /* Second call for description. */
UF_CALL( UF_ATTR_assign( atteid, "DESCRIPTN", att_value ) );
```

Assign attributes. Please note that `att_value.value.string` is a pointer variable. It is not an allocated character array. When you want to assign a string-valued attribute, you set up the string by assigning `att_value.value.string` to a character array. `svalue` is our character array. The string can be up to 132 characters (not including the NULL character).

The assignment of `att_value.value.string` to `svalue` needs to be done only once. Once the part number attribute has been assigned, it takes less code to assign the description attribute.

`att_value` is our attribute data structure. Please note that it is declared with the typedef `UF_ATTR_value_t` NOT with `UF_ATTR_value_p_t`.

Because we are using one part per solid body, it makes sense to assign attributes to the part. `UF_ATTR_ask_part_attribute(&atteid)` gets the tag of the part attribute entity.

TIP Only strings can be assigned to the part attribute. Integer and other types of attributes can be assigned to geometric objects.

The code segment discussed can be used to assign part attributes to all three parts by changing the `svalue` string to the appropriate value in the bottom and button part creation routines.

calcx.c

The external Open API program is written as a single code mainline segment. It can be compiled with the same switches as an internal Open API program. It does not harm anything for this to be compiled as position independent. The link phase for external Open API programs will require specifying 'e' for external rather than the default of 'i' for internal.

Code Discussion: calcx.c

```

/* Check for correct number of input arguments */
  if(argc != 3) {
    strcpy(msg,"This program takes two arguments: ");
    strcat(msg,"the root part file name and the name\n" );
    strcat(msg,"of the output text file.");
    printf("%s\n",msg);
    exit(1);
  }

```

The program assumes two arguments. argv[0] is the program name (occurs for all C programs), the first argument argv[1] is the part file name, and argv[2] is the output text file. This code protects the program from an improper argument count.

```

/* Check out a Open API license */
  flag = UF_CALL( UF_initialize() );
  if(flag) exit(2);

```

External Open API application programs must get a license for Open API function calls to be made.

```

/* Open the part. It should be argv[1]. Store its tag as top_part. */
  strcpy(fspect, argv[1] );
  flag = UF_CALL( UF_PART_open(fspect,&top_part,&estat) );
  if(flag) exit(3);

```

Open the part name from the calling arguments. It should be the name of your calculator part. Copy the calling argument into a local variable (fspect) that is large enough for the maximum file specification length.

```

/* Call the routine to set the view to a model view (if a drawing is
 * displayed). It has no effect if a model view is active.
 */
  UF_DRAW_set_display_state(1);

```

As part of this program, we are going to be changing the work part. This code sets the drawing display state off. As with interactive Unigraphics, you cannot change the work part in an assembly if the drawing is displayed. UF_DRAW_set_display_state will insure that a model view is 'active'.

```

/* Open the text file (use fid for file identifier). */
  fid = uc4504(argv[2],3,0);
  if(fid < 0) {
    UF_get_fail_message(flag,msg);
    printf("%s\n",msg);
    exit(4);
  }
/* Write a header "Report for calculator" to the file id. */
  uc4524(fid,"Report for Calculator");
  uc4524(fid,"-----");

```



Create a text file for writing. **argv[2]** is the second argument to the program and is the name of the file to create. The **3** tells *uc4504* to delete any existing copy of the text file. The third argument to *uc4504* is the file type. Unigraphics uses this to check for the suffix on the file name. For instance, if you were working with parts, you would specify a file type of **2**. Unigraphics would use this to make sure that a “.prt” is included in the file name.

Here, we are using a file type of **0** which tells *uc4504* not to worry about the suffix. We will rely on the user to know the name of the file.

```
/* Set the attribute char pointer to point to the svalue character string
   array. Initialize the savename to a NULL string. */
att_value.value.string = svalue;
strcpy(savename, "");
```

The structure member **att_value.value.string** is a pointer variable. Memory must be allocated or it must be set to point to an existing character array.

The **savename** variable will be used to keep the program from reporting on every instance of the button. Only the first instance will be reported.

```
/* Loop through all the component instances. For each unique instance,
 * obtain the attribute information and write it to the file.
 */
inst = NULL_TAG;
while( TRUE ){

/* Set inst to the next instance under the top_part using the assem
 * routine to cycle instances. Get the part names of the instances.
 */
inst = UF_ASSEM_cycle_inst_of_part(top_part,inst);
if( NULL_TAG == inst )
    break;
UF_ASSEM_ask_part_name_of_child(inst,fspec);

/* Split the file spec into the directory and part name strings. Remove
 * the '.prt' from the part file string and save the result in pname.
 */
uc4576 (fspec,2,dspec,cname);
uc4578 (cname,2,pname);

/* Check if pname is the same as the previous name. Skip it if it is (the
 * buttons will have the same name and should only be done once.
 */
if (strcmp(savename,pname) == 0) {
    continue;
}
printf("Working on %s\n",pname);
```

Cycle through the instances inside the calculator part. For each instance, get its name and if the name is not already in **savename**, get the child part of the instance and make it the work part.

UF_ASSEM_ask_part_name_of_child returns the full filespec of the part. *uc4576* splits a filespec into the directory path and the file name. The file type (2) is used to insure the name has the “.prt” suffix. *uc4578* strips the suffix from a file name.

```

/* Write a blank line and then the file name. */
    uc4524(fid," ");
    uc4524(fid,pname);

/* Set child to the child part tag of the instance and change
 * the work part to the child so we can get the part attribute.
 * Check return code.
 */
    child = UF_ASSEM_ask_child_of_instance(inst);
    flag = UF_CALL( UF_ASSEM_set_work_part(child) );
    if(flag) exit(5);

/* Get the part attribute ID and read the PARTNUMBER and DESCRIPTN
 * attribute values.
 */
    UF_CALL( UF_ATTR_ask_part_attribute(&atteid) );
    UF_CALL( UF_ATTR_read_value(atteid,"PARTNUMBER",UF_ATTR_string,
        &att_value) );
    sprintf(msg,"    Part Number = %s",svalue);
    uc4524(fid,msg);

    UF_ATTR_read_value(atteid,"DESCRIPTN",UF_ATTR_string,
        &att_value);
    sprintf(msg,"    Description = %s",svalue);
    uc4524(fid,msg);

```

Then get the part attribute entity and read the attributes. Build a string, **printf(msg," Part Number = %s",svalue)**, with three spaces, the attribute title, and the attribute value. Write the string to the text file (*uc4524*).

```

/* Copy the name of the part we just did to the save string */
    strcpy(savename,pname);
};

```

Put the name of the part into savename to avoid doing the button parts multiple times. This is the last statement in the while loop that cycles the instances

```

/* Close and print the file */
    uc4540(fid,0);
    sprintf(msg,"cat %s,argv[1]");
    system(msg);

```



The file is closed and saved to the file system. Then the file is displayed.

```
/* Release Open API license */
  UF_terminate();
  exit(0);
}
```

Release the Open API license.

Activity: Attribute and External Open API

Step 1 Add attributes to the top, bottom, and button parts. Edit files *calc_model_top.c*, *calc_model_bottom.c* and *calc_model_button.c*. Add the Open API function calls to add part attributes to the part files at the comments marked **YYY**.

Step 2 Copy the template file *calcx.c* to your sub-directory.

Step 3 Add the appropriate Open API calls.

NOTE On windows systems, change the system command *cat* to the windows dos command *type*.

Step 4 Create a make file for the external Open program. Use the external variable names in the make file. You will not have any SUBOBS. Name the file *ext_calc* because you already have a file called *Makefile* in your directory. Use “make ext -f ext_calc” to create your executable.

Step 5 Run the internal Open API application program *calculator*. This time, you must save the calculator part files, which now have number and description attributes.

Step 6 Run the external Open API application program. To run the program:

- enter the program file name followed by:
- the part file and
- an arbitrary data file name.
E.g. *calcx calculator.prt calc_att.dat*

NOTE On Windows systems, make sure that *libufun.dll* is in the current path. This file is found in \$UGII_ROOT_DIR. Start, Programs, Unigraphics, Unigraphics Tools, UG Command Prompt.



SUMMARY

You added part attributes to our calculator model and saved your assembly. You then created an external Open API application program that opened the assembly, read the attributes and generated a report.

In this lesson, you :

- Learned how to make an external Open API application program.
- Learned how to create attributes on parts and objects.
- Read attributes from part files.
- Used a program with CFI (Common File Interface) calls.

Plotting

Appendix A



Plotting in the Open API

Another step in the calculator project is plotting the drawing. *calc_plot* is the function that does this. It sets up the plotting options and uses the function `UF_PLOT_drawing`.

Plot Routines

The routines to plot in the Open API can plot either a drawing or the current display. These routines are available in both internal and external Open API.

- `UF_PLOT_display`
- `UF_PLOT_drawing`

Code Discussion: *calc_plot.c*

```
UF_PLOT_options_t plot_options;
```

`UF_PLOT_options_t` is a typedef for the structure shown earlier. It creates the structure that will be passed to the plot routine.

```
/* Set the plot options. The scale of .45 allows an A2 to plot on 8.5X11
 * paper. The rotation will and offset will be zero. The origin is the
 * lower left corner, 0,0,0.
 */
    plot_options.scale = 0.45;
```

To plot this A2 size drawing to an 8 1/2 x 11 inch sheet, use a scale of 0.45.

```
plot_options.plotter = pltr;
plot_options.node    = node;
plot_options.jobname = jobnm;
plot_options.pauseMsg = pause;
plot_options.bannerMsg = banner;
```

The character arrays are not allocated. Set the structure character pointer variables equal to the variable names we declared as character arrays.

```
/* Call the plot drawing routine. Check return code. */
flag = UF_PLOT_drawing(dname, &plot_options, &jobid);
if(flag < 0) return(flag);
```

Call the plot routine with the drawing name argument passed to the routine.

Activity: Generating Drawing Plots



- Step 1** Copy the template file *calc_plot.c* to your sub-directory.

- Step 2** Add the Open API function calls to plot the drawing. Make sure the destination plotter name is valid!

- Step 3** Add the call to *calc_plot* in the *calculator* program.

- Step 4** Add the prototype for *calc_plot* in the *calcproto.h* file.

- Step 5** Edit *Makefile* and add *calc_plot.o* to the SUBOBJS list and make a new executable. When you run the program, you should get a plot.



(This Page Intentionally Left Blank)

Glossary

ABS – Absolute coordinate system.

Absolute Coordinate System – Coordinate system in which all geometry is located from a fixed or absolute zero point.

active view – One of up to 49 views per layout in which you can directly work.

angle – In Unigraphics, an angle measured on the X-Y plane of a coordinate system is positive if the direction that it is swept is counterclockwise as viewed from the positive Z axis side of the X-Y plane. An angle swept in the opposite direction is said to be negative.

arc – An incomplete circle; sometimes used interchangeably with the term “circle.”

ASCII – American Standard Code for Information Interchange. It is a set of 8-bit binary numbers representing the alphabet, punctuation, numerals, and other special symbols used in text representation and communications protocol.

aspect ratio – The ratio of length to height which represents the change in size of a symbol from its original.

assembly – A collection of piece parts and sub-assemblies representing a product. In Unigraphics, an assembly is a part file which contains components.

assembly part – A Unigraphics part file which is a user-defined, structured combination of sub-assemblies, components and/or objects.

associativity – The ability to tie together (link) separate pieces of information to aid in automating the design, drafting, and manufacture of parts in Unigraphics.

attributes – Pieces of information that can be associated with Unigraphics geometry and parts such as assigning a name to an object.

block font – A Unigraphics character font which is the default font used for creating text in drafting objects and dimensions.

body – Class of objects containing sheets and solids (see solid body and sheet body).

bottom-up modeling – Modeling technique where component parts are designed and edited in isolation of their usage within some higher level assembly. All assemblies using the component are automatically updated when opened to reflect the geometric edits made at the piece part level.



boundary – A set of geometric objects that describes the containment of a part from a vantage point.

CAD/CAM – Computer Aided Design/Computer Aided Manufacturing.

category, layer – A name assigned to a layer, or group of layers. A category, if descriptive of the type of data found on the layers to which it is assigned, will assist the user in identifying and managing data in a part file.

chaining – A method of selecting a sequence of curves which are joined end-to-end.

circle – A complete and closed arc, sometimes used interchangeably with the term “arc.”

component – A collection of objects, similar to a group, in an assembly part. A component may be a sub-assembly consisting of other, lower level components.

component part – The part file or “master” pointed to by a component within an assembly. The actual geometry is stored in the component part and referenced, not copied, by the assembly. A separate Unigraphics part file that the system associates with a component object in the assembly part.

cone direction – Defines the cone direction using the Vector Subfunction.

cone origin – Defines the base origin using the Point Subfunction.

half angle – The half vertex angle defines the angle formed by the axis of the cone and its side.

constraints – Refer to the methods you can use to refine and limit your sketch. The methods of constraining a sketch are geometric and dimensional.

construction points – Points used to create a spline. Construction points may be used as poles (control vertices), defining points, or data points. See POLES, DEFINING POINTS, and DATA POINTS.

control point – Represents a specific location on an existing object. A line has three control points: both end points and the midpoint of the line. The control point for a closed circle is its center, while the control points for an open arc are its end and midpoints. A spline has a control point at each knot point. A control point is a position on existing geometry. Any of the following points: 1. Existing Points 2. Endpoints of conics 3. Endpoints and midpoints of open arcs 4. Center points of circles 5. Midpoints and endpoints of lines 6. Endpoints of splines.

convert curve – A method of creating a b-curve in which curves (lines, arcs, conics or splines) may be selected for conversion into a b-curve.

Coordinate System – A system of axes used in specifying positions (CSYS).

counterclockwise – The right-hand rule determines the counter-clockwise direction. If the thumb is aligned with the ZC axis and pointing in the positive direction, counterclockwise is defined as the direction the fingers would move from the positive XC axis to the positive YC axis.

current layout – The layout currently displayed on the screen. Layout data is kept in an intermediate storage area until it is saved.

curve – A curve in Unigraphics is any line, arc, conic, spline or b-curve. A geometric object; this may refer to a line, an arc, a conic, or a spline.

defaults – Assumed values when they are not specifically defined.

defining points – Spline construction points. Splines created using defining points are forced to pass through the points. These points are guaranteed to be on the spline.

degree-of-freedom arrows – Arrow-like indicators that show areas that require more information to fully constrain a sketch.

design in context – The ability to directly edit component geometry as it is displayed in the assembly. Geometry from other components can be selected to aid in the modeling. Also referred to as edit in place.

dimensional constraint – This is a scalar value or expression which limits the measure of some geometric object such as the length of a line, the radius of an arc, or the distance between two points.

directory – A hierarchical file organization structure which contains a list of filenames together with information for locating those files.

displayed part – The part currently displayed in the graphics window.

edit in place – See design in context.

emphasize work part – A color coding option which helps distinguish geometry in the work part from geometry in other parts within the same assembly.

endpoint – An endpoint of a curve or an existing point.

expression – An arithmetic or conditional statement that has a value. Expressions are used to control dimensions and the relationships between dimensions of a model.

face – A region on the outside of a body enclosed by edges.



feature – An all-encompassing term which refers to all solids, bodies, and primitives.

file – A group or unit of logically related data which is labeled or “named” and associated with a specified space. In Unigraphics, parts, and patterns are a few types of files.

filtering – See object filtering.

font box – A rectangle or “box” composed of dashed line objects. The font box defines the size, width and spacing of characters belonging to a particular font.

font, character – A set of characters designed at a certain size, width and spacing.

font, line – Various styles of lines and curves, such as solid, dashed, etc.

free form feature – A body of zero thickness. (see body and sheet body)

generator curve – A contiguous set of curves, either open or closed, that can be swept or revolved to create a body.

geometric constraint – A relationship between one or more geometric objects that forces a limitation. For example, two lines that are perpendicular or parallel specifies a geometric constraint.

grid – A rectangular array of implied points used to accurately align locations which are entered by using the “screen position” option.

guide curve – A set of contiguous curves that define a path for a sweep operation.

virtual intersection – Intersection formed by extending two line segments that do not touch to the position that they cross. The line segments must be non-parallel and coplanar.

inflection – A point on a spline where the curve changes from concave to convex, or vice versa.

interactive step – An individual menu in a sequence of menus used in performing a Unigraphics function.

isometric view (Tfr-ISO) – Isometric view orientation – one where equal distances along the coordinate axes are also equal to the view plane. One of the axes is vertical.

knot points – The defining points of a spline. Points along a B-spline, representing the endpoints of each spline segment.

layer – A layer is a partition of a part. Layers are analogous to the transparent material used by conventional designers. For example, the user may create all geometry on one layer, all text and dimensions on a second, and tool paths on a third.

layout – A collection of viewports or window areas, in which views are displayed. The standard layouts in Unigraphics include one, two, four or six viewports.

layouts – Standard layouts are available to the user. These include:

L1 – Single View,

L2 – Two Views,

L3 – Two Views,

L4 – Four Views,

L6 – Six Views.

Information window – The window used in listing operations, such as **Info**.

loaded part – Any part currently opened and in memory. Parts are loaded explicitly using the *File*→*Open* option and implicitly when they are used in an assembly being opened.

menu – A list of options from which the user makes a selection.

model space – The coordinate system of a newly created part. This is also referred to as the “absolute coordinate system.” Any other coordinate system may be thought of as a rotation and/or translation of the absolute coordinate system.

name, expression – – The name of an expression is the single variable on the left hand side of the expression. All expression names must be unique in a part file. Each expression can have only one name. See expression.

objects – All geometry within the Unigraphics environment.

offset face – A Unigraphics surface type created by projecting (offsetting) points along all the normals of a selected surface at a specified distance to create a “true” offset.

options – A number of various alternatives (functions, modes, parameters, etc.) from among which the user can choose.

origin – The point $X = 0, Y = 0, Z = 0$ for any particular coordinate system.

parametric design – Concept used to define and control the relationships between the features of a model. Concept where the features of the model are defined by parameters.

part – A Unigraphics file containing a .prt extension. It may be a piece part containing model geometry, a sub-assembly, or a top-level assembly.



part or model – A collection of Unigraphics objects which together may represent some object or structure.

partially loaded part – A component part which, for performance reasons, has not been fully loaded. Only those portions of the component part necessary to render the higher level assembly are initially loaded (the reference set).

point set – A distribution of points on a curve between two bounding points on that curve.

Point Subfunction Menu – A list of options (methods) by which positions can be specified in Unigraphics.

read-only part – A part for which the user does not have write access privilege.

real time dynamics – Produces smooth pan, zoom, and rotation of a part, though placing great demand on the CPU.

Refresh – A function which causes the system to refresh the display list on the viewing screen. This removes temporary display items and fills in holes left by *Blank* or *Delete*.

right-hand rule, conventional – The right-hand rule is used to determine the orientation of a coordinate system. If the origin of the coordinate system is in the palm of the right fist, with the back of the hand lying on a table, the outward extension of the index finger corresponds to the positive Y axis, the upward extension of the middle finger corresponds to the positive Z axis, and the outward extension of the thumb corresponds to the positive X axis.

right-hand rule for rotation – The right-hand rule for rotation is used to associate vectors with directions of rotation. When the thumb is extended and aligned with a given vector, the curled fingers determine the associated direction of rotation. Conversely, when the curled fingers are held so as to indicate a given direction of rotation, the extended thumb determines the associated vector.

screen cursor (cursor) – A marker on the screen which the user moves around using some position indicator device. Used for indicating positions, selecting objects, etc. Takes the form of a full-screen cross.

sheet – A object consisting of one or more faces not enclosing a volume. A body of zero-thickness. Also called sheet body.)

sketch – A collection of geometric objects that closely approximates the outline of a particular design. You refine your sketch with dimensional and geometric constraints until you achieve a precise representation of your design. The sketch can then be extruded or revolved to obtain a 3D object or feature.

Sketch Coordinate System (SCS) – The SCS is a coordinate system which corresponds to the plane of the sketch. When a sketch is created the WCS is changed to the SCS of the new sketch.

solid body – An enclosed volume. A type of body (see Body).

spline – A smooth free-form curve.

stored layout – The last saved version of a layout.

stored view – The last saved version of a view.

string – A contiguous series of lines and/or arcs connected at their end points.

sub-assembly – A part which both contains components and is itself used as a component in higher-level assemblies.

surface – The underlying geometry used to define a face on a sheet body. A surface is always a sheet but a sheet is not necessarily a surface (see sheet body). The underlying geometry used to define the shape of a face on a sheet.

system – The Unigraphics System.

temporary part – An empty part which is optionally created for any component parts which cannot be found in the process of opening an assembly.

top-down modeling – Modeling technique where component parts can be created and edited while working at the assembly level. Geometric changes made at the assembly level are automatically reflected in the individual component part when saved.

trim – To shorten or extend a curve.

trimetric view (Tfr-Tri) – A viewing orientation which provides you with an excellent view of the principal axes. In Unigraphics II, the trimetric view has the Z-axis vertical. The measure along the X-axis is $\frac{7}{8}$ of the measure along Z, and the measure along the Y-axis is $\frac{3}{4}$ of the measure along Z.

Unigraphics – A computer based turnkey graphics system for computer-aided design, drafting, and manufacturing, produced by UGS.

units – The unit of measure in which you may work when constructing in Unigraphics. Upon log on, you may define the unit of measure as inches or millimeters.

upgraded component – A component which was originally created pre-V10 but has been opened in V10 and upgraded to remove the duplicate geometry.



version – A term which identifies the state of a part with respect to a series of modifications that have been made to the part since its creation.

view – A particular display of the model. View parameters include view orientation matrix; center; scale; X,Y and Z clipping bounds; perspective vector; drawing reference point and scale. Eight standard views are available to the user: Top, Front, Right, Left, Bottom, Back, Tfr-ISO (top-front-right isometric), and Tfr-Tri (top-front-right trimetric).

view dependent edit – A mode in which the user can edit a part in the current work view only.

view dependent modifications – Modifications to the display of geometry in a particular view. These include erase from view and modify color, font and width.

view dependent geometry – Geometry created within a particular view. It will only be displayed in that view.

WCS – Work Coordinate System.

WCS, work plane – The WCS (Work Coordinate System) is the coordinate system singled out by the user for use in construction, verification, etc. The coordinates of the WCS are called work coordinates and are denoted by XC, YC, ZC. The XC-YC plane is called the work plane.

Work Coordinate System – See WCS.

work layer – The layer on which geometry is being constructed. You may create objects on only one layer at a time.

work part – The part in which you create and edit geometry. The work part can be your displayed part or any component part which is contained in your displayed assembly part. When displaying a piece part, the work part is always the same as the displayed part.

work view – The view in which work is being performed. When the creation mode is view dependent, any construction and view dependent editing that is performed will occur only in the current work view.

XC axis – X-axis of the work coordinate system.

YC axis – Y-axis of the work coordinate system.

ZC axis – Z-axis of the work coordinate system.

Index

A

ABS, GL-1
 Absolute Coordinate System, GL-1
 Active View, GL-1
 Angle, GL-1
 Arc, GL-1
 ASCII, GL-1
 Aspect Ratio, GL-1
 Assemblies, GL-1
 Associativity, GL-1
 Attribute, GL-1

B

Body, GL-1
 Bottom-Up Modeling, GL-1
 Boundary, GL-2

C

Category, Layer, GL-2
 Chaining, GL-2
 Circle, GL-2
 Component, GL-2
 Part, GL-2
 Cone
 Direction, GL-2
 Origin, GL-2
 Constraints, GL-2
 Construction Points, GL-2
 Control Point, GL-2
 Convert, Curves to B-Curves, GL-2
 Coordinate Systems, GL-3
 Sketch, GL-7
 Counterclockwise, GL-3
 Current Layout, GL-3
 Cursor, GL-6
 Curve, GL-3

D

Defaults, GL-3
 Defining Points, GL-3
 Degree-of-freedom Arrows, GL-3
 Design in Context, GL-3
 Dimension Constraints, GL-3
 Direction, Cone, GL-2
 Directory, GL-3
 Displayed Part, GL-3

E

Edit in Place, GL-3
 Emphasize Work Part, GL-3
 Endpoint, GL-3
 Expressions, GL-3
 Names, GL-5

F

Face, GL-3
 Features, GL-4
 File, GL-4
 Filtering, GL-4
 Font
 Box, GL-4
 Character, GL-4
 Line, GL-4
 Free Form Feature, GL-4

G

Generator Curve, GL-4
 Geometric Constraint, GL-4
 Grid, GL-4
 Guide Curve, GL-4

H

Half Angle, GL-2



I

Inflection, GL-4

K

Knot Points, GL-4

L

Layer, GL-5

Layout, GL-5

Listing Window, GL-5

Loaded Part, GL-5

M

Menu, GL-5

Model, GL-6

Model Space, GL-5

O

Object, GL-5

Offset Surface, GL-5

Origin, Cone, GL-2

P

Parametric Design, GL-5

Part, GL-5, GL-6

Partially Loaded Part, GL-6

Point Set, GL-6

Point Subfunction, GL-6

R

Read-Only Part, GL-6

Real Time Dynamics, GL-6

Refresh, GL-6

Right Hand Rule, GL-6

Rotation, GL-6

S

SCS, GL-7

Sheet, GL-6

Sketch, GL-6

Coordinate System, GL-7

Solid Body, GL-7

Spline, GL-7

Stored Layout, GL-7

Stored View, GL-7

String, GL-7

Sub-assembly, GL-7

Surface, GL-7

System, GL-7

T

Temporary Part, GL-7

Tfr-ISO, GL-4

Tfr-Tri, GL-7

Top-Down Modeling, GL-7

Trim, GL-7

U

uc5540, 6-2

uc6430, 6-13

uc6431, 6-13

uc6432, 6-13

uc6433, 6-13

uc6434, 6-13

uc6442, 6-13

uc6443, 6-13

uc6449, 6-13

uc6450, 6-13

uc6454, 6-13

uc6464, 6-13

uc6466, 6-13

uc6468, 6-13

uc6476, 7-3

uc6477, 7-3

uc6478, 7-3

uc6479, 7-3



- uc6480, 7–3
- uc6481, 7–3
- uc6482, 7–3
- uc6483, 7–3
- uc6484, 7–3
- uc6485, 7–3
- uc6486, 7–3
- uc6492, 7–3
- uc6495, 7–3
- uc6496, 7–3
- uc6499, 7–3
- UF_ASSEM_ask_assem_options, 3–11
- UF_ASSEM_ask_child_of_instance, 3–11
- UF_ASSEM_ask_component_data, 3–11
- UF_ASSEM_ask_inst_of_part_occ, 3–11
- UF_ASSEM_ask_mc_array_data, 3–11
- UF_ASSEM_ask_occs_of_entity, 3–11
- UF_ASSEM_ask_occs_of_part, 3–11
- UF_ASSEM_ask_parent_of_instance, 3–11
- UF_ASSEM_ask_prototype_of_occ, 3–11
- UF_ASSEM_ask_work_part, 3–11
- UF_ASSEM_create_component_part, 3–11
- UF_ASSEM_create_mc_array, 3–11
- UF_ASSEM_cycle_ents_in_part_occ, 3–11
- UF_ASSEM_is_occurrence, 3–11
- UF_ASSEM_is_part_occurrence, 3–11
- UF_ASSEM_remove_instance, 3–11
- UF_ASSEM_set_assem_options, 3–11
- UF_ATTR_ask_part_attribute, 8–3
- UF_ATTR_ask_part_attrs, 8–3
- UF_ATTR_ask_part_attrs_in_file, 8–3
- UF_ATTR_assign, 8–3
- UF_ATTR_cycle, 8–3
- UF_ATTR_delete, 8–3
- UF_ATTR_delete_all, 8–3
- UF_ATTR_read_value, 8–3
- UF_CSYS_ask_wcs, 6–12
- UF_CSYS_create_csys, 6–12
- UF_CSYS_create_matrix, 6–12
- UF_CSYS_create_temp_csys, 6–12
- UF_CSYS_map_point, 6–12
- UF_CSYS_set_wcs, 6–12
- UF_DRAW_ask_current_drawing, 7–3
- UF_DRAW_ask_drawing_info, 7–3
- UF_DRAW_define_view_manual_rect, 7–3
- UF_DRAW_import_view, 7–3
- UF_DRAW_set_drawing_info, 7–3
- UF_DRAW_update_one_view, 7–3
- UF_DRF_ask_preferences, 6–2
- UF_DRF_create_horizontal_dim, 6–2
- UF_DRF_create_label, 6–2
- UF_DRF_create_vertical_dim, 6–2
- UF_DRF_init_object_structure, 6–2
- UF_DRF_set_preferences, 6–2
- UF_LAYER_ask_status, 3–12
- UF_LAYER_cycle_by_layer, 3–12
- UF_LAYER_set_status, 3–12
- UF_MODL_active_part, 5–9
- UF_MODL_ask_block_parms, 5–10
- UF_MODL_ask_body_edges, 5–10
- UF_MODL_ask_body_faces, 5–10
- UF_MODL_ask_body_type, 5–10
- UF_MODL_ask_edge_body, 5–10
- UF_MODL_ask_edge_faces, 5–10
- UF_MODL_ask_edge_type, 5–10
- UF_MODL_ask_edge_verts, 5–10
- UF_MODL_ask_exp, 4–4
- UF_MODL_ask_exp_tag_string, 4–4
- UF_MODL_ask_exp_tag_value, 4–4
- UF_MODL_ask_face_body, 5–10
- UF_MODL_ask_face_data, 5–10
- UF_MODL_ask_face_edges, 5–10
- UF_MODL_ask_feat_body, 5–10
- UF_MODL_ask_feat_faces, 5–10
- UF_MODL_ask_list_count, 5–3
- UF_MODL_ask_list_item, 5–3
- UF_MODL_ask_minimum_dist, 5–10
- UF_MODL_ask_simple_hole_parms, 5–10
- UF_MODL_create_blend, 5–9
- UF_MODL_create_block1, 5–9
- UF_MODL_create_cyl1, 5–9



UF_MODL_create_exp, 4–4
UF_MODL_create_exp_tag, 4–4
UF_MODL_create_hollow, 5–9
UF_MODL_create_linear_iset, 5–9
UF_MODL_create_list, 5–3
UF_MODL_create_rect_pocket, 5–9
UF_MODL_create_rect_slot, 5–9
UF_MODL_create_simple_hole, 5–9
UF_MODL_delete_exp, 4–4
UF_MODL_delete_exp_tag, 4–4
UF_MODL_delete_list, 5–3
UF_MODL_delete_list_item, 5–3
UF_MODL_dissect_exp_string, 4–4
UF_MODL_edit_exp, 4–4
UF_MODL_eval_exp, 4–4
UF_MODL_export_exp, 4–4
UF_MODL_import_exp, 4–4
UF_MODL_operations, 5–9
UF_MODL_put_list_item, 5–3
UF_MODL_rename_exp, 4–4
UF_MODL_update, 4–4
UF_OBJ_ask_name, 5–11
UF_OBJ_ask_status, 3–13
UF_OBJ_ask_type_and_subtype, Object, 3–13
UF_OBJ_cycle_by_name, 5–11
UF_OBJ_delete_name, 5–11
UF_OBJ_delete_object, 3–13
UF_OBJ_set_def_cre_color, 3–13
UF_OBJ_set_name, 5–11
UF_PART_import, 7–4
UF_PLOT_display, A–1
UF_PLOT_drawing, A–1
UF_VIEW_ask_tag_of_view_name, 6–2
UF_VIEW_ask_work_view, 6–13

UF_VIEW_cycle_objects, 6–13
UF_VIEW_delete, 6–13
UF_VIEW_expand_view, 6–2
UF_VIEW_fit_view, 6–13
UF_VIEW_is_expanded, 6–2
UF_view_save_all_active_views, 6–13
UF_VIEW_unexpand_work_view, 6–2
Unigraphics, GL–7
Units, GL–7
Upgrade, Component, GL–7

V

Version, GL–8
View, GL–8
 Isometric, GL–4
 Trimetric, GL–7
 Work, GL–8

W

WCS, GL–8
Work Layer, GL–8
Work Part, GL–8

X

XC-Axis, GL–8

Y

YC-Axis, GL–8

Z

ZC-Axis, GL–8

