

**DEVELOPMENT OF AIRCRAFT FLIGHT INSTRUMENTS GRAPHICS
DISPLAY SOFTWARE IN THE *EIFFEL* OBJECT-ORIENTED
ENVIRONMENT**

Eric Trenk

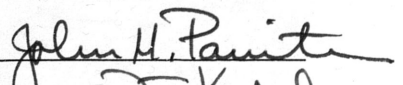
University Undergraduate Fellow 1990-91

Texas A&M University

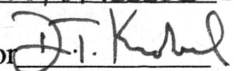
Department of Electrical Engineering

APPROVED:

Felows Advisor



Honors Program Director



ABSTRACT

**Development of Aircraft Flight Instruments Graphics Display Software in the Eiffel
Object-oriented Environment. (April 1991)**

Eric Trenk, University Undergraduate Fellow

Advisor: Dr. John H. Painter

This thesis reports the results of research to develop, using the Eiffel object-oriented programming language, aircraft flight instruments display software to graphically display output from a passenger jet transport simulation. The display will be used in the development of a knowledge-based flight mode interpreter and controller for the aircraft simulation. The results of this project reinforce Eiffel's suitability for graphical applications.

TABLE OF CONTENTS

	Page
I INTRODUCTION.....	1
II BRIEF OVERVIEW OF OBJECT-ORIENTED PROGRAMMING.....	3
A. Tenets of Object-oriented Programming.....	3
B. Advantages over procedure based programming.....	7
III BRIEF OVERVIEW OF THE EIFFEL PROGRAMMING LANGUAGE.....	10
A. Reasons for selecting Eiffel.....	10
B. Key concepts of the Eiffel Language.....	12
IV DESIGN OF AIRCRAFT FLIGHT INSTRUMENTS DISPLAY SOFTWARE..	16
A. Specifications.....	16
B. Approach.....	18
C. Design Issues.....	21
V CONCLUSIONS AND RECOMMENDATIONS.....	25
A. Review.....	25
B. Remarks.....	25
C. Future work.....	26
REFERENCES.....	27
APPENDIX.....	30
A. Sample Eiffel source code.....	30
B. Sample C intermediate source code.....	32

LIST OF FIGURES

Figure	Page
1 Flow Diagram: Knowledge-Based Management for Dynamic Systems.....	2
2 Eiffel map metaphor.....	15
3 User interface class relationships.....	20
4 Software gauge displays.....	23

I INTRODUCTION

"Object-oriented" has replaced "structured" as the high tech adjective for "good" software design [1]. For software development within this new programming paradigm, a variety of programming languages and environments have emerged as possible standards. This research is an attempt to evaluate the suitability of one of these environments, the Eiffel object-oriented programming language, for graphical applications such as a display of aircraft flight instruments. The key questions about Eiffel hoped to be answered by this investigation include the following:

- 1) ease of learning the language - how long will someone with moderate experience in procedure based programming require to become adept at Eiffel?
- 2) speed of coding - after learning Eiffel, how long will be required to produce a given program?
- 3) efficiency of code - how will a completed Eiffel program compare in size and execution time to a program with equivalent functions written in another language?
- 4) readability of code - how readily will another programmer be able to determine the function of an Eiffel program?
- 5) extendibility/adaptability to future requirements - how easily will past Eiffel programs be able to be adapted to new needs?

The aircraft flight instruments display was developed as part of a software package for research in the automatic control of dynamic systems. The display will be used in conjunction with a passenger jet transport simulation, a knowledge-based flight mode interpreter, and a knowledge-based aircraft controller (see figure 1). Ideally, all parts of the

system will run concurrently and share data in real time. Except for the simulation, which is written in the C language, all parts are to be written in Eiffel. This display research will also determine to some extent the suitability of Eiffel to non-graphical applications such as the flight mode interpreter and the controller.

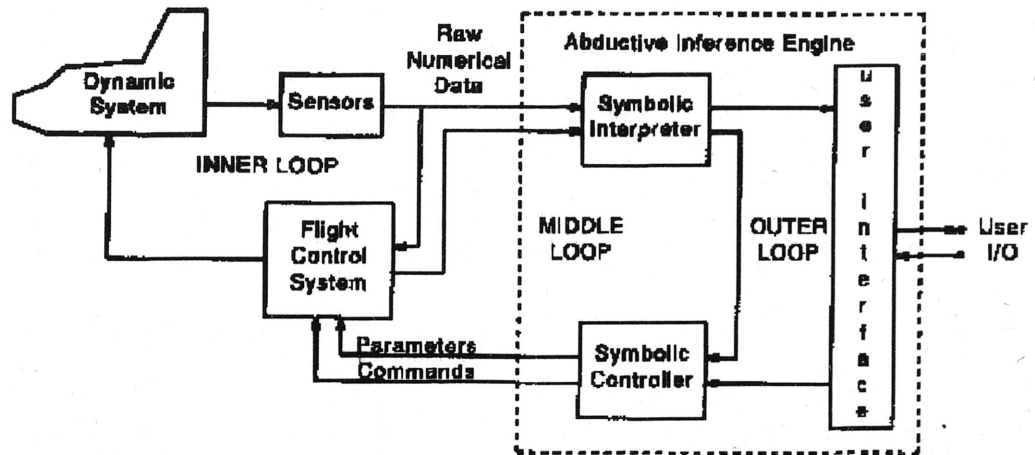


Figure 1. Top Level Data-Flow Diagram
 Knowledge-Based Management for Dynamic Systems
 (Figure courtesy of J.H. Painter [1])

II BRIEF BACKGROUND OF OBJECT-ORIENTED PROGRAMMING

Object-oriented does not refer to a programming language; it refers to programming concepts that can be implemented through new object-oriented languages or through most existing languages [2]. In recent years, this approach has received a great deal of attention, especially from those interested in graphical applications. An overview of object oriented programming is appropriate before a discussion of the application of this technique to particular display software.

A. Tenets of Object-oriented Programming

Although the idea of object-oriented programming has been around for twenty years, the specifics and terminology are still not fixed (see [3] for more on standardizing object-oriented terminology). This section will discuss in general terms the principles common to most object-oriented implementations.

1. Classes and Objects

Predictably, the object is the basic building block of object-oriented programming. Objects are often chosen to simulate real world objects, such as a plant, a bank account, or a graphics window. These objects include all of the relevant data plus all of the functions and routines that may be applied to that data. This combination of data and functions makes an object more than a data structure from a procedure-based program.

In most languages, objects are instances of classes. A class can be thought of as a specification for an object, but note that the object does not exist until it is explicitly created. The class is a list of features, which can be attributes or procedures or functions. In comparison with traditional programming, a class can be thought of as a type definition and an object as a variable declaration [4].

2. Encapsulation

To take full advantage of object-orientedness, the objects must be as self-contained and independent of their environment as possible. The idea of autonomous constructs is not new (it is a basic principle of structured programming); what is new is the idea of self-contained units containing both data and procedures [2]. Organizing programs into objects containing data that is generally not shared with other objects is known as encapsulation or information hiding.

Encapsulation provides many benefits, especially in large, complex systems. The software is simplified by reducing the amount of interaction between modules (objects). Also, the specification of a module is separated from its implementation, reducing the impact of later changes to the implementation method.

3. Inheritance

Inheritance provides an organizational structure for classes. The structure is hierarchical, like the biological classification system which starts at the very general (Kingdom) and becomes progressively more specific (Subkingdom, Phylum, ... ,Species), or, even more similarly, like the directory tree structure of DOS or UNIX which begins with a general root directory that may contain files or subdirectories containing

progressively more specific files or subdirectories. With inheritance, a class may be instantiated and/or a more specific class can become an “heir” to it. This heir class inherits all attributes, procedures, and routines, just as if the text of the previous class were copied directly into it. The heir is then free to add or modify features as needed. This saves the programmer from duplicating code unnecessarily.

As an example, the Eiffel libraries provide a class FIGURE, which contains features such as display, which is common to all graphical figures [5]. Classes for specific figures such as CIRCLE or RECTANGLE inherit from class FIGURE and make use of these general features. The more specific class will then add features which are unique to that class, such as “draw,” which is implemented differently for a circle than for a rectangle.

Inheritance minimizes the amount of new code needed when classes are added to a system by providing a simple method to take advantage of existing code. Inheritance also simplifies the debugging process because a modification to a single general class will affect all of the classes that inherit from it. There are some problems associated with inheritance, however. Name conflicts can arise if the name of a feature in the more general class is also used in the heir class. Conflicts can also arise because the data type created by the heir class does not exactly conform to that of the general class.

4. Polymorphism

Polymorphism means “many shapes” in Greek, and in object-oriented terms means the ability for different objects to respond to the same message or function call in different ways. This individualized response relieves a procedure from the responsibility of knowing what kind of data it is dealing with, therefore allowing more generic and reusable code. Using the previous example of graphical figures, a procedure could execute

the call “A.draw,” where, if taking advantage of polymorphism, “A” could refer to a circle or a rectangle. The draw procedure for a circle would be different than that for a rectangle, but the proper method would be implemented for the reference of “A” at the time of the call. The procedure making the call “A.draw” does not need to know the correct implementation.

Closely tied to the concept of polymorphism are the concepts of typing and binding. The degree of typing a language allows determines to what extent it allows polymorphism. A statically typed language demands that an object retains the same data type from compile-time throughout run-time. The validity of calls to the object are checked during compilation. Because the data type is always the same, polymorphism can not exist. A weakly typed language allows full flexibility with data types; that is, little or no type checking occurs during compilation. This method allows full use of polymorphism, but can result in run-time errors since a feature can be called on an object (data type) that does not have that feature. For example, if “A” is declared of type integer (continuing the previous example), “A.draw” would not be valid. A weakly typed language may not catch this error during compilation, but a statically typed language would.

Eiffel is one language that provides a compromise between these two typing styles: strong typing. A strongly typed language does some type checking during compilation, but also allows polymorphism. In Eiffel, the type checking restriction is as follows:

If the type used to declare an entity is called its static type, and the type of the object associated with it at some run-time instant is called its dynamic type, the type rule expresses that the dynamic type must be a descendant of the static type [6].

If the dynamic type inherits (is a descendant of) the static type, it logically has all of the features associated with the static type and should not produce a run-time error due to a call to an unavailable feature. Actually, this type rule does not preclude run-time errors caused by type discrepancies (see [7] for a discussion of this Eiffel flaw). Strength of typing is

important in the versatility of a language, but presents a design compromise between flexibility and robustness.

Binding is a term associated with typing. Early binding means that calls to a variable feature are fixed at compile-time because the variable's data type is fixed (statically typed). Late or dynamic binding means that calls can not be fixed and must be made during run-time since the data type of a variable can change. Early binding produces no execution time overhead, while late binding does result in some performance loss. Again, this issue is a language design compromise.

B. Advantages over procedure based programming

Brad Cox, in [8], compares the current paradigm for software development to the Aristotelian cosmological model. This model of the universe placed the earth at the center, with the sun, moon, and planets circling around in spheres. To account for observed discrepancies in planetary motion, Ptolemy began to add epicycles to these spheres. By the sixteenth century, 90 epicycles were needed, resulting in enough complexity for the Pope to ask Copernicus to investigate the matter. Copernicus's solution was a change of paradigm-- a universe with planets rotating around the sun.

The software revolution, Cox argues, will be an equivalent change of paradigm, from procedure-based to object-based. This change will similarly allow a simplification of the complexity that has evolved over time.

This analogy may overstate the magnitude of object-oriented programming's impact on software design, but is in general consistent with the opinion of many scholars and industry experts. Software lacks the modularity and reusability found in mechanical parts, electrical components, or even computer hardware. In order to keep up with the increasing demand for quality code, the software industry must reassess its development methods.

Eiffel author Bertrand Meyer expresses his strong views on the need for a restructuring of software development practices in [9].

Object-oriented programming may not be the final solution to problems in the software industry, but it does provide advantages over traditional programming styles.

1. Ease of design

Traditional top-down software design philosophy mandates that a task be defined at its most general level, then divided into subtasks, divided further, and so on until the modules are specific enough to code. Object-oriented design rejects this philosophy for several reasons. A task is never completely specified, so to define it in its most general terms is not possible. In reality, the focus of a project may shift after coding begins or even after it is finished. Some modules may also require additional features after the design is specified. Top-down design does not accommodate this possibility well.

Object-oriented programming does not demand that the final structure be frozen before coding begins. Any anticipated need for a class can be filled before its relationship to the rest of the system is fully specified. Additional features can be added to a class at any time, without any impact to other classes in the system. This ability allows for the rapid prototyping of a system, since skeletons of classes can be written and later filled in. Object-oriented design is not necessarily bottom-up, as the class relationships must be well planned. This planning, however, does not later limit the flexibility of the code.

Object-oriented programming has been shown in some cases to provide a productivity gain over procedure-based programming with respect to the size of code [10], though code length is not guaranteed to be shortened and may in fact increase. Old objects can (and ideally should) be reused in new systems, however, so code does not have to be written from scratch. This reuse usually results in faster development and fewer bugs.

2. Ease of Maintenance

It has been estimated that seventy percent of the cost of any software system is due to maintenance [9]. One of the goals of object-oriented programming is to make this maintenance easier and therefore less expensive. Maintenance of software usually includes fixing bugs, adapting to new formats, and modifying to fit revised user requirements. All of these activities are made easier by using object-oriented techniques.

Errors are easier to trace when dealing with classes not because the program flow is simpler (it is definitely not), but because there is less interaction between classes. Thanks to encapsulation, a problem will remain more localized and easier to find than in a system where the same data is touched by dozens of modules. Also, the complexity of code is controlled by the forced use of a hierarchical structure, making the program more easily read and understood. And for modifications, inheritance provides a perfect tool to make small additions without jeopardizing the existing program and without writing extensive new code.

III BRIEF BACKGROUND OF THE EIFFEL PROGRAMMING LANGUAGE

The Eiffel language was designed by Bertrand Meyer and his group at Interactive Software Engineering [6]. Currently, Interactive Software is the prime resource for Eiffel products, but the language specification is in the public domain and anyone is permitted to write compilers, interpreters, tools, or libraries for Eiffel. The International Eiffel Consortium was founded to take full control over the evolution of the language, and Interactive plans to relinquish the Eiffel trademark to the consortium.

A. Reasons for using Eiffel

There are about a dozen object-oriented languages commonly in use today [11]. Choosing the proper one for a given application can be a difficult matter, since there are many factors to compare. Included here is some explanation for why Eiffel was chosen for this project.

1. Object-oriented factors

Choosing an object-oriented language implies choosing among the language design trade-offs discussed previously, such as strength of typing and flexibility of inheritance. Some languages are heavy on object-orientedness at the expense of clear structure while other offer the opposite. Eiffel provides somewhat of a compromise when compared to other popular languages.

C++ and Objective-C are object-oriented languages that are built as extensions to standard C. They offer the possibility of writing procedure-based routines that can coexist with objects when (if) that is advantageous. Unfortunately, this ability relaxes the programmer's responsibility to write very object-oriented code. Especially for newcomers to the object-oriented paradigm, this freedom fosters bad habits. Languages built object-oriented from the ground up, like Smalltalk or Eiffel, force stricter compliance with the underlying principles of object-oriented programming.

Many languages, such as Smalltalk, Actor, Objective-C, and Object Pascal do not support multiple inheritance (therefore a class may be the heir of only one class). This restriction limits design creativity and code reusability. Some languages like Ada do not support any inheritance or dynamic binding. Eiffel seems quite powerful in comparison. Eiffel allows unlimited inheritance, strong typing (providing polymorphism but also some type checking), and dynamic binding.

2. Other factors

Although not an extension of C, Eiffel allows easy interfacing with C or other existing languages. Eiffel supports call-out of utilities written in other languages and call-in of Eiffel routines from those other languages [6]. Eiffel also incorporates the ability to interface with X-Windows, a feature important for graphical applications. Eiffel comes with extensive libraries including 300 classes totaling about 5000 accessible features [6],[9]. These libraries will probably form the foundation for any software system written in Eiffel. Some other positive aspects are that the Eiffel environment includes graphical browsing and debugging tools, Eiffel code is very readable, and finally, although all object-oriented languages suffer from a lack of good documentation, Eiffel has (in addition

to user's manuals) a companion book written by the author of the language, which explains the motivation for and correct implementation of Eiffel constructs.

B. Key concepts of the Eiffel language

The principle reference for programming theory for Eiffel is Bertrand Meyer's *Object-oriented Software Construction* [9]. For a thorough treatment of this theory the Meyer source is suggested. This section will attempt only to identify briefly some of the key concepts of Eiffel.

1. Guidelines

Meyer has identified several factors that influence the quality of any software. His assertion is that object-oriented programming, particularly in Eiffel, encourages quality code. The factors are:

correctness - ability to perform specified tasks

robustness - ability to function in abnormal conditions

extendibility - ease with which may be adapted to changes in specifications

reusability - ability to be reused for new applications

compatibility - ease with which may be combined with others.

Also identified as significant are efficiency, portability, verifiability, integrity, and ease of use. This research did not intend to investigate the importance of these listed factors. They are presented here because they are an integral part of working in the Eiffel environment, and when considered, provide good checks on the quality of written code.

2. Class relationships

Classes in Eiffel can have only one of two relationships: ancestor/descendent or client/supplier. Anytime a class must be aware of another class's presence, the decision of which type of relationship to use must be made. Being a client means accessing features through the official interface, while being an heir means having access to the implementation of the features and having the ability to redefine or adapt class properties [6]. A client is given less authority and flexibility, but is more insulated from errors in or future changes in implementation in other classes. Usually a class is made a client of another class unless it meets the "is a" criterion. A class should only inherit if it "is a" whatever the ancestor class is. For example, a POLICEMAN class should inherit the PERSON class, but should be a client of the BADGE class. A policeman needs a badge, but is not one.

Classes that are intended for inheritance only (will never be instantiated) are explicitly tagged "deferred" in Eiffel. These deferred classes have at least one deferred feature; that is, a feature that is named but whose implementation is not set. The implementation for this deferred feature is left to the heir class to define. In the aircraft flight instruments graphics display software, both client/supplier and ancestor/descendent relationships were used, as well as deferred classes.

3. Variable sharing

Generally in Eiffel, a feature is not available anywhere outside the class it belongs to unless absolutely necessary. This is to protect both the feature and the potential user of the feature (refer to the concept of encapsulation). In an Eiffel class, features that are to be

made available to the outside are listed in the "export" clause, so a feature that is not exported can not be accessed or seen by another class (unless the other class inherits it, in which case nothing is hidden).

Within a given class, there is also encapsulation. Each feature can declare local variables, meaning other features even of the same class can not access them. It is tempting to view this another layer of information hiding within the class level information hiding; however, it is not exactly the same. The value of a non-exported variable (attribute) declared within a class does not change value unless the change is made explicitly, but a variable declared as local to a feature (routine or procedure) is re-instantiated every time the feature is entered. Before it was understood, this difference caused significant problems with the display software. A sound guideline for design is to declare outside a feature (that is, not declare as local) any variable that needs to maintain its value after the feature is called or between occurrences of the feature being called.

There is no Eiffel language construct for global variables or the creation of objects that need to be visible to all classes. The method used in the display software for effectively creating global objects is discussed in the display design section of this work.

4. Graphics "map" metaphor

Implementing graphics in Eiffel is analogous to manipulating a geographical map (see [5] for more on Eiffel graphics). All figures, in this case gauges, are drawn on a "world." First the world is created (it is possible to create more than one world), then the graphical figures are attached. The world is two-dimensional, extends infinitely, and has scale not tied to that of the device or window. Next a window is created to look at part of the world. The window has a size on the display and a separate size coverage area on the world. This

allows implicit scaling of figures from world size to display size. Figure 2 portrays this map metaphor.

For this display, subwindows were established and tied to a common root or parent window. Subwindows are assigned a display size and location just like independent windows, and are only distinguishable from an independent window because of an "attach" command to tie them to their parent. The portion of the world displayed in a subwindow is in no way related to the subwindow's position.

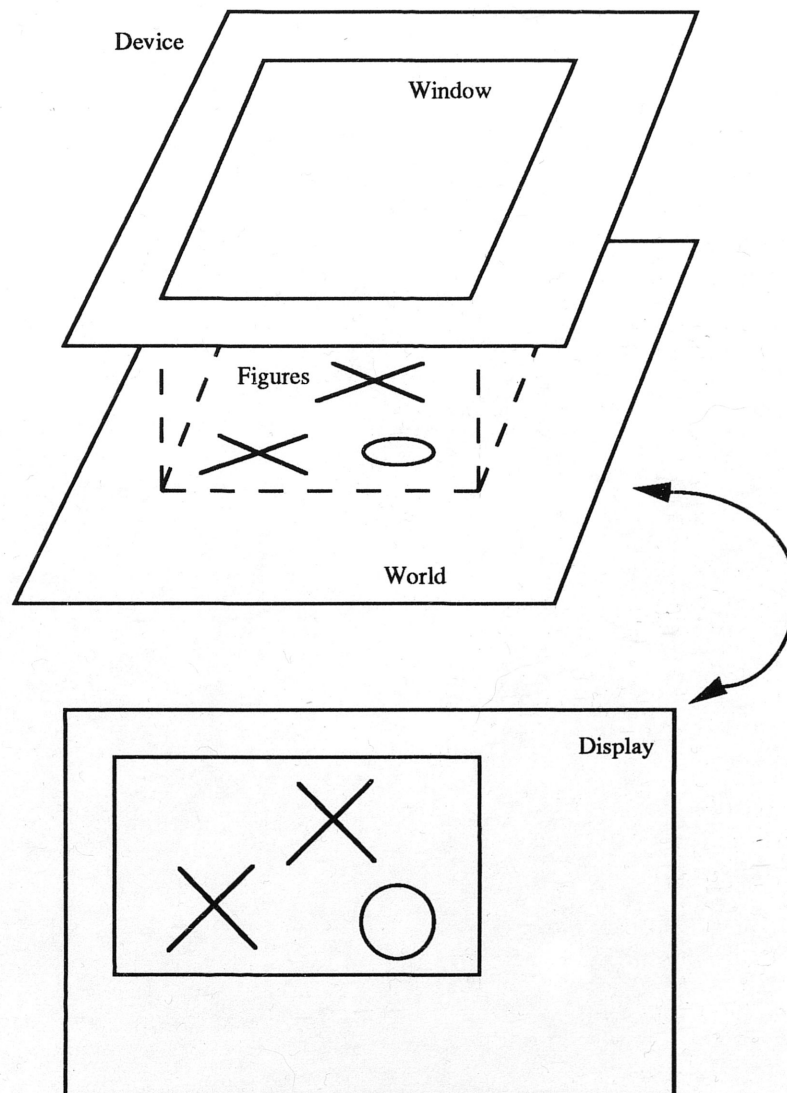


Figure 2. Eiffel graphics "map" metaphor

IV DESIGN OF AIRCRAFT FLIGHT INSTRUMENTS DISPLAY SOFTWARE

The increasing popularity and complexity of graphical user interfaces has sparked interest in object-oriented programming for this application [12],[13]. This research project is one example of the use of the object-oriented paradigm for graphics.

A. Specifications

The requirements for the aircraft flight instruments display were loosely defined, partially because it was to be used as part of a system that was still under development and not fully specified itself, and to an even greater extent because the capabilities of Eiffel were not completely known (to determine these capabilities was one of the primary motivations for the project).

1. Appearance

The goal was to create a display as similar as possible in gauge layout and appearance to that of a transport jet, within the constraints of reasonable hardware and software. Since none of the navigational gauges that are prominent in modern instrument panels were used, the display could not really conform to the real world model. Also many of the details of the gauges were modified for programming efficiency whenever the extra code required exceeded the benefit of a more faithful reproduction. For example, all of the display gauges were created with linearly proportional scales, even though many of the actual

gauges have graduated scaling. The time and distance (DME) gauges were replaced with simple numeric displays also because the extra detail was deemed unnecessary.

2. Interprocess Communication

The display was designed to be supplied with data from the aircraft simulation running on the workstation concurrently. With the current release of Eiffel, concurrent processes are not supported (concurrent processes are expected in a later release), so the system was designed to alternate between simulation processing and graphics display processing. Eiffel has incorporated the ability to interface with C, so from the instruments display side, the only requirement was to provide a feature which accepts eight real numbers (representing the values of eight gauges), and updates the display with these new values if appropriate. This feature is provided in the root class of the instruments display software.

3. Code Complexity

The aircraft flight instruments display was also constrained by processing power. The efficiency of a graphics program written in Eiffel was not known, but for this application the display code certainly could not require a "disproportionate" amount of CPU time. Eiffel is very structured from the programmer's viewpoint, but the Eiffel code is compiled to C as an intermediate step before being compiled to an executable form. This intermediate step allows for portability to a variety of hardware platforms, but it also introduces inefficiency in the implementation of Eiffel commands. The C code generated is so complex that the Eiffel manuals [5] discourage the programmer from even looking at his own program once it is compiled to C (a portion of this code is included as Appendix B).

The final user interface consisted of 122 Eiffel classes which were compiled to 122 C listings which were then linked together. The display performance was improved considerably by eliminating assertion checking, a feature of Eiffel which provides ongoing verification of the code's correctness. This reduced the size of the executable code and decreased the display time by a factor of about three. Assertion checking is disabled by setting the appropriate compilation option in the Eiffel system description file.

The initialization of the display requires about two seconds when running exclusively on a Sun 386i workstation. Time for updating the gauges, the key speed consideration, is yet to be determined for the fully operating system, but appears to be around one half to one second. Processing demands can usually be met with faster hardware, and this solution was attempted by porting the code to a Sun SparcStation (14 Mips vs. 3.5 Mips for the 386i). The speed improvement was much less than expected, indicating that the delay was not primarily from the Eiffel code, but from the X-Windows interface. The delay in updating the display is such a critical issue that it may prevent the current implementation from being considered a long term solution. Again, determining this type of information was a primary purpose of this research.

B. Approach

1. Objects

To design any object-oriented program, the first step is to find the objects. For this project the primary objects were gauges -- not gauges in the physical sense of a circle with a needle, but in an abstract sense of an entity that displays a value. This definition was necessary since the display window was designed with six square subdivisions, but with eight separate values to be displayed. As a result, more than one value or "gauge" was to

be displayed in some of the physical gauge sites. For example, the Mach number is passed from the simulation separately from the air speed, even though they are related (the relationship is based on temperature, which is not passed from the simulation, so one can not be derived from the other by the display code). The Mach number is displayed as a two digit number on the dial face of the airspeed gauge, but exists as a separate gauge-- a parallel, not subordinate, entity to the airspeed gauge.

2. Structure

The instrument display was constructed with these eight gauges as the primary objects. Ideally, they could be eight "instances" of the same gauge class since they were all the same "type" of object. This would be the equivalent of running the same procedure eight times with different input data. The differences between the gauges were so numerous, however, this tactic was judged not feasible. It should be noted, however, that an equivalent display could be achieved in this manner. It would require a single gauge class containing every feature any of the instances of gauge required. Then the root class could pass every parameter needed when it created the instance, including location, size, title, start and end values, and tick placement value information.

A more modular approach was taken by using inheritance. All of the features of the gauges that were common (to at least two of the eight) were made part of a base class called GAUGE. Then a class was written for each of the more specific gauge types, such as the altitude gauge, airspeed gauge, and artificial horizon gauge. The more specific gauge types inherited the general class GAUGE and therefore contained all the primary features of a "gauge." Any feature specific to a particular gauge type was then added only to that type, reducing the number of unnecessary features in some of the classes. This implementation took advantage of the commonalties between the gauges, but also allowed for unlimited

differences. A root class (called GUI) was established as a client of the individual gauge classes, and it is from this class that the calls to create actual gauge objects are made. No parameters need to be passed from the root class for initialization, and additional features can be added later to one or all of the gauges without changing the interface between the root class and the individual gauge classes. In fact the root class need not be aware of any changes. This is entirely consistent with the principle of modularity basic to object oriented programming. Figure 3 shows the relationships between the principle Eiffel classes of the flight instruments display software.

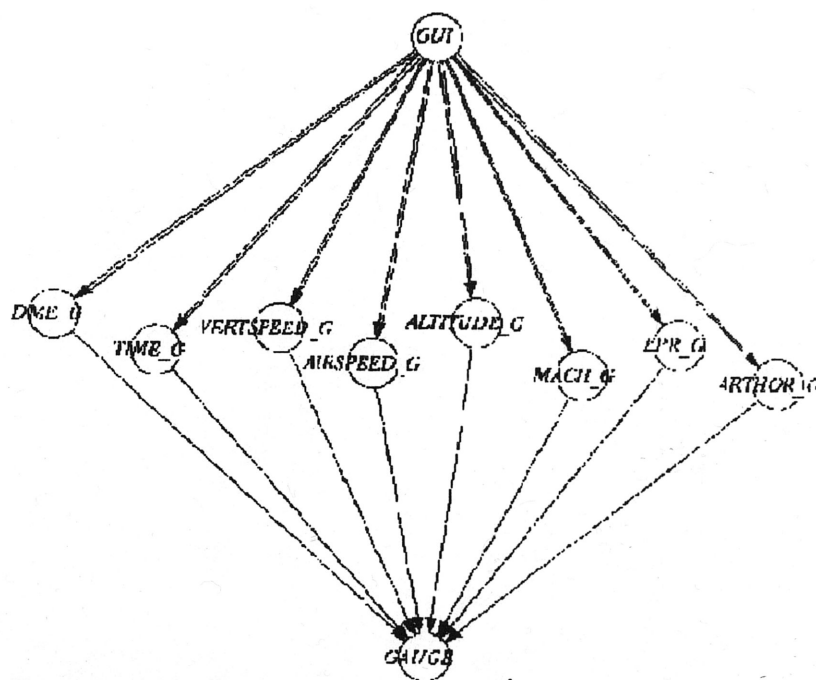


Figure 3. Display software class relationships

C. Design Issues

1. Global Variables

As has been mentioned previously, Eiffel has no facility for declaring global variables [14]. This was a conscious design decision, made because global variables are not consistent with encapsulation. In some instances, however, global variables are necessary (actually, global variables are never absolutely necessary, but can be considered necessary when compared to the complexity of the alternatives in some cases). In the display software, the same world needed to be used by each of the individual gauge classes as well as the root class. With no modifications to the code, the world class could be accessed by all of them, but they would not be dealing with the same instance of the class (object). The solution presented by Meyer (see [14] for details) is to create the global objects in a single class (can be called GLOBAL_OBJ), and inherit GLOBAL_OBJ into every class that needs the objects. The GLOBAL_OBJ class is not shown on the class diagram in figure 3, but it is inherited by both GUI and GAUGE, and so is seen by all of the classes (it is seen by ALTITUDE_G for example because ALTITUDE_G inherits GAUGE). This method of achieving global variables, although not entirely straightforward, worked without error.

2. Window refresh

An unexpected problem arose when the gauges were actually updated on the screen; the old needle positions were displayed in addition to the new positions. This was surprising since there existed only one needle object per gauge in the code, and this object was rotated to update the displayed value. It seemed impossible to have more than one

needle on the screen. The problem, it is now known, arises from the Eiffel to X-Windows interface and the X-Windows method for maintaining graphics on the screen. Pixels are not reset automatically (it is assumed that this would be too computationally burdensome), so to refresh the screen, it has to be explicitly filled with white. The current version of the display software fills with white all subwindows containing a gauge that has been updated, then redisplay the window. This method is effective but slow. Another alternative would be to draw white over the previous needle anytime a new needle is drawn.

Figure 4a shows the aircraft flight instruments graphics display; figure 4b shows the display as it appears on the workstation screen under X-Windows.

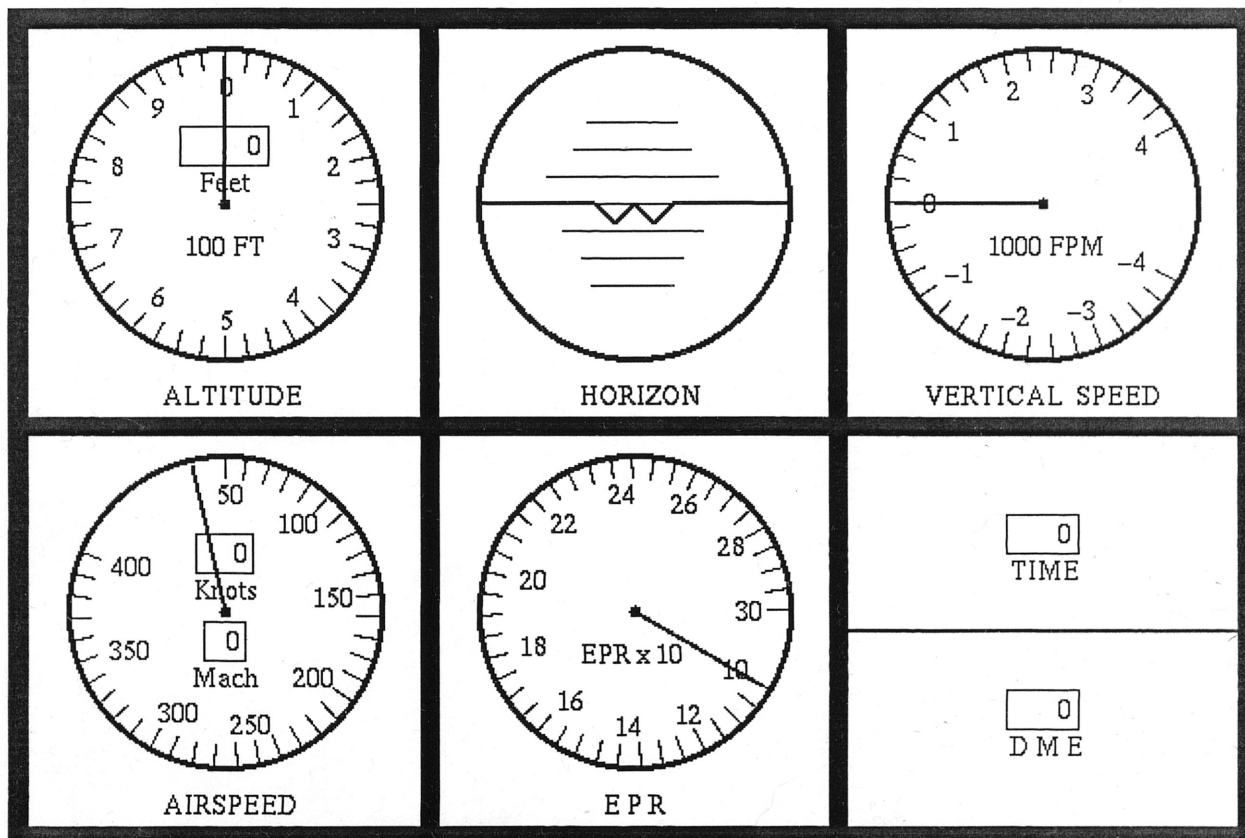


Figure 4a. Gauge display

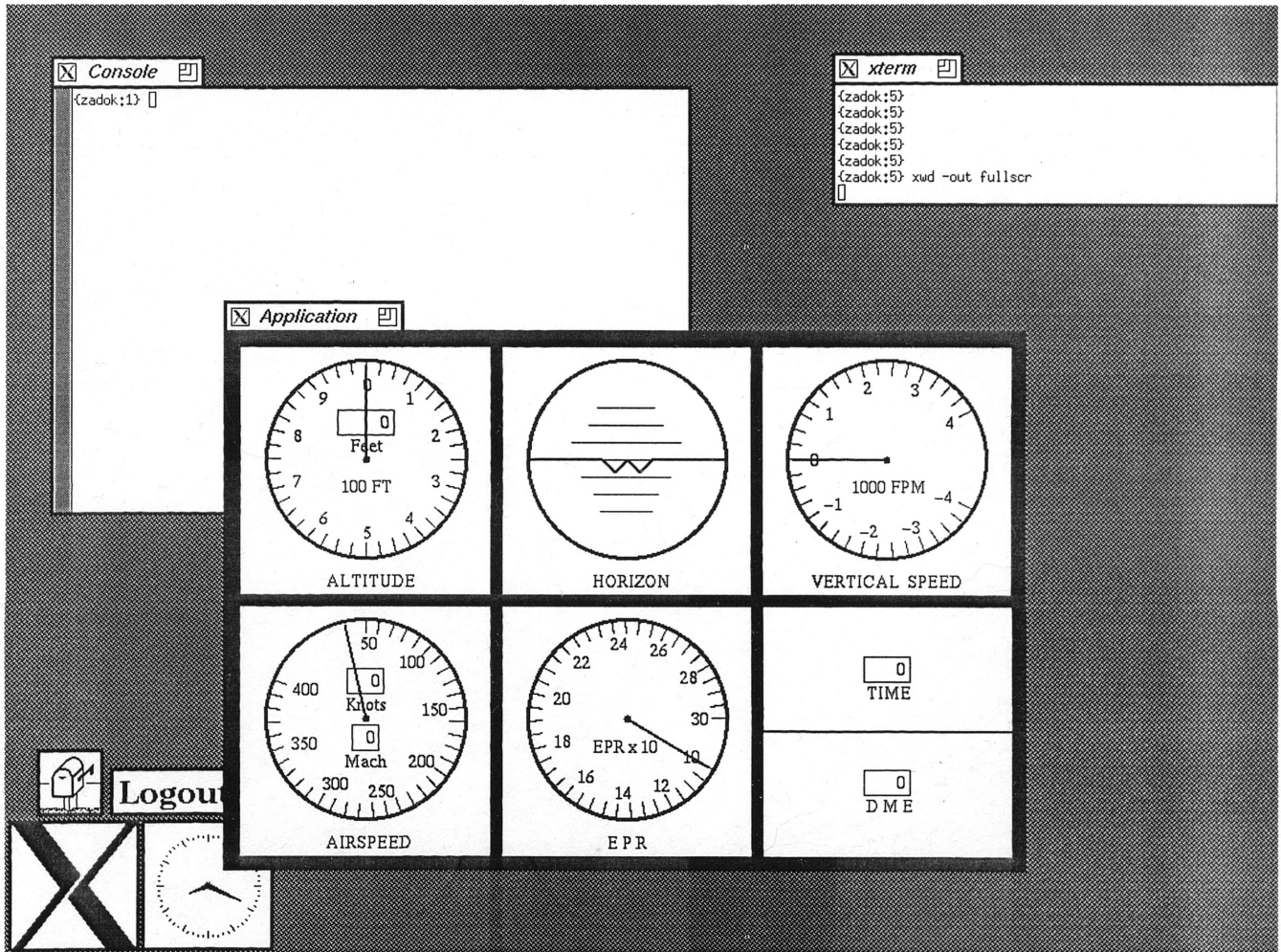


Figure 4b. Workstation monitor with gauge display

V CONCLUSIONS AND RECOMMENDATIONS

A. Review

Object-oriented programming represents a shift in software design and development paradigm. Although there are many different languages available, the underlying principles of object-oriented programming remain the same. They include encapsulation, inheritance, and polymorphism and result in increased reliability, adaptability, and extensibility of software. To create a flight instruments graphics display, the specifications were established, the approach and class structure determined, and the major design problems were overcome. From this exercise a better understanding of object-oriented programming was established, and object-oriented programming in general and Eiffel in particular were determined to be overall well suited for this type of application. This conclusion is not unqualified, however, as code performance issues have not been resolved and the large learning curve associated with object-oriented programming has not been unequivocally justified.

B. Remarks

Learning object-oriented programming is more difficult than simply learning a new language. Object-oriented programming requires a fundamental change in thinking [2]. The magnitude of this learning curve is one of the major results of this research, as its size was originally underestimated. It may in fact deter future forays into object-oriented programming except for applications that absolutely demand this method.

This research was also hampered by other problems, including workstation bugs not associated with Eiffel, bugs in Eiffel still in existence because of the newness of the language, and lack of Eiffel documentation also attributable to newness. These problems were not in general preventable and probably must be endured by any other researchers following this path.

C. Future Work

Many questions still remain unanswered at the conclusion of this work. This display software can possibly be optimized now that more is known about the behavior of Eiffel. Also, comparisons can be made to similar applications written in other object-oriented languages and even non-object-oriented languages. Object-oriented programming is still evolving, and its suitability for many types of software projects is still to be determined.

REFERENCES

- [1] J. H. Painter, "Knowledge-Based Management of Dynamic Systems," *Research Presentation*, Department of Electrical Engineering, Texas A&M University, February 26, 1991.
- [2] G. S. Howard, "Object Oriented Programming Explained," *Journal of Systems Management*, vol. 39, pp. 13-19, July 1990.
- [3] R. J. Wirfs-Brock and R. E. Johnson, "Surveying Current Research in Object-Oriented design," *Communications of the ACM*, vol. 33, no. 9, pp.104-123, Sept. 1990.
- [4] D. Thomas, "What's in an Object?," *Byte*, vol. 14, pp. 231-240, Mar. 1989.
- [5] B. Meyer and J. M. Nelson, *Eiffel: The Libraries*, CA: Interactive Software Engineering, 1990.
- [6] B. Meyer, "Lessons from the Design of the Eiffel Libraries," *Communications of the ACM*, vol. 33, no. 9, pp. 69-87, Sept. 1990.
- [7] W. R. Cook, "A Proposal for Making Eiffel Type-safe," *The Computer Journal*, vol. 32, no. 4, pp. 305-311, Aug. 1989.
- [8] B. J. Cox, "There Is a Silver Bullet," *Byte*, vol. 15, pp. 209-218, Oct. 1990.
- [9] B. Meyer, *Object-oriented Software Construction*, Englewood Cliffs, NJ: Prentice Hall, 1988.
- [10] F. S. Kuhl, "Object-oriented Programming Applied to a Prototype Workstation," *Software: Practice and Experience*, vol. 20, no. 9, pp. 887-898, Sept. 1990.
- [11] T. Manuel, "Object-oriented programming: a star is born," *Electronics*, vol. 62, pp. 104-108, May 1989.
- [12] Z. Urlocker, "Object-Oriented Programming for Windows," *Byte*, vol. 15, pp. 287-294, May 1990.
- [13] D. Pountain, "Object-Oriented Programming," *Byte*, vol. 15, pp. 257-264, Feb. 1990.
- [14] B. Meyer, "Bidding Farewell to Globals," *Journal of Object-oriented Programming*, vol. 1, no. 3, pp. 73-76, Aug/Sept 1988.

SUPPLEMENTAL REFERENCES CONSULTED

- [1] D. G. Bobrow, "The Object of Desire," *Datamation*, vol. 35, pp. 37-41, May 1, 1989.
- [2] S. Wilson, "The object-oriented approach," *IEE Review*, vol. 36, pp. 277-280, July/August 1990.
- [3] J. Duntemann and C. Marinacci, "New Objects for old Structures," *Byte*, vol. 15, pp. 261-266, April 1990.
- [4] B. Meyer, "Eiffel: A Language and Environment for Software Engineering," *IEEE Software*, vol. 4, no.2, pp. 50-64, Mar. 1987.
- [5] L. V. Mancini, "A Technique for Subclassing and its Implementation Exploiting Polymorphic Procedures," *Software: Practice and Experience*, vol.18, pp. 287-300, April 1988.
- [6] D. G. Kafura and K. H. Lee, "Inheritance in Actor Based Concurrent Object-Oriented Languages," *The Computer Journal*, vol. 32, no. 4, pp. 297-304, Aug. 1989.
- [7] M. A. Linton, J. M. Vlissides, and P. R. Calder, "Composing User Interfaces with InterViews," *Computer*, vol. 22, pp. 8-22, Feb. 1989.
- [8] P. Brownlow, "Get a handle on object-oriented programming," *EDN*, vol..35, pp. 265-276, Nov. 8, 1990.
- [9] C. Duff and B. Howard, "Migration Patterns," *Byte*, vol. 15, pp. 223-232, Oct. 1990.
- [10] E. Gibson, "Objects--Born and Bred," *Byte*, vol. 15, pp.245-254, Oct. 1990.
- [11] E. Yourdon, "Auld Lang Syne," *Byte*, vol. 15, pp. 257-264, Oct. 1990.
- [13] P. Wegner, "Learning the Language," *Byte*, vol. 14, pp.245-253, Mar. 1989.
- [14] M. H. Dodani, C. E. Hughes, and J.M. Moshell, "Separation of Powers," *Byte*, vol. 14, pp.254-262, Mar. 1989.
- [15] J. P. Jacky and I. J. Kalet, "An Object-Oriented Programming Discipline for Standard Pascal," *Communications of the ACM*, vol. 30, no. 9, pp. 772-776, Sept. 1987.

- [16] T. Korson and J. D. McGregor, "Understanding Object-Oriented: A Unifying Paradigm," *Communications of the ACM*, vol. 33, no. 9, pp. 40-60, Sept. 1990.
- [17] D. Jordan, "Implementation Benefits of C++ Language Mechanisms," *Communications of the ACM*, vol. 33, no. 9, pp.61-64, Sept. 1990.
- [18] G. Agha, "Concurrent Object-Oriented Programming," *Communications of the ACM*, vol. 33, no. 9, pp. 125-141, Sept. 1990.
- [19] B. Meyer, "From Structured Programming to Object-Oriented design: The Road to Eiffel," *Structured Programming*, vol. 10, no. 1, pp. 19-39, 1989.
- [20] *American Airlines 737 Operating Manual*, American Airlines Flight Department, Ft. Worth, TX, 1988.

ACKNOWLEDGEMENTS

The author would like to thank Dr. John H. Painter, research advisor for this project, for his support and assistance, and Greg Economides, fellow researcher, for the benefit of his knowledge of the Eiffel environment.

APPENDIX A

Sample Eiffel Source Code

The following is the Eiffel source code for one class of the display software. This class is titled "ALTITUDE_G" and represents the specification for the altitude gauge. Most of the features called are inherited from the more general class GAUGE. Note how this inheritance allows the class to be more abstract and readable.

```

class ALTITUDE_G export
  set_value
inherit
  GAUGE;
feature
  Create is
    do
      center.Create(100.0,90.0);
      radius:=80;
      -- set parameters for dial face
      draw_dial;

      title_len:=8;
      title.Create(8);
      title.set("ALTITUDE",1,8);
      -- set parameters for displaying title
      display_title;

      units_len:=5;
      units.Create(6);
      units.set("100 FT",1,6);
      -- set parameters for gauge units display
      display_units;

      start_val:=0.0;
      end_val:=10.0;
      start_pos:=0.0;
      end_pos:=2*pi;
      start_label:=0.0;
      end_label:=9.0;
      label_inc:=1.0;
      hash_inc:=0.25;
      -- set parameters for hash marks and
      -- hash mark labels
      draw_hash;

      box_label_len:=3;
      box_label.Create(4);
      box_label.set("Feet",1,4);
      -- set parameters for numeric value display box

      value_power:=100.0;
      value_len:=5;
      set_value(0.0);
      -- set parameters for display of
      -- gauge value (needle & box)
    end; -- Create

  set_value (value: REAL) is
    do
      set_needle(value);
      set_box(value);
    end; -- set_value

end; --class ALTITUDE_G

```

APPENDIX B

Sample C Intermediate Source Code

The following is the intermediate source code for the Eiffel class ALTITUDE_G. The Eiffel class is first compiled to this form, then to object code for execution. Note the difficulty in determining the function of this code.

```

#include "_eiffel.h"

extern int16   *D02A, *DD02A; /*{ 02A altitude_g */
extern void    Dive02B (); /*{ 02B gauge */
extern int16   *D02B, *DD02B; /*{ 02B gauge */
extern int16   *D002, *DD002; /*{ 002 string */
DATUM   _02A000_altitude_g_create ();
extern DATUM   _02B001_gauge_draw_dial (); /* gauge draw_dial */
extern DATUM   _00200h_string_set (); /* string set */
extern DATUM   _02B004_gauge_draw_hash (); /* gauge draw_hash */
extern DATUM   _02B003_gauge_display_units (); /* gauge display_units */
extern DATUM   _02B002_gauge_display_title (); /* gauge display_title */
DATUM   _02A001_altitude_g_set_value ();
extern DATUM   _02B007_gauge_set_needle (); /* gauge set_needle */
extern DATUM   _02B008_gauge_set_box (); /* gauge set_box */
void    keep02A ();
/*% altitude_g 02A 2 2 */

static int class= 160 ;
/*% create */DATUM
_02A000_altitude_g_create (BCurrent)
OBJPTR BCurrent;
{
    static int routine = -1;
    GACDEC1
    TRACEDEC
    DATUM *DCurrent;
    float *FCurrent;
    OBJPTR t1;
    int16   type;
    GACIN1;
    TRACEIN;
    SETJMP2;
    _CR_INVARIANT;
    _C_INVARIANT;
    DCurrent=Access(BCurrent)+/*@ *//D02B[DType(BCurrent)];
    FCurrent=(float *)DCurrent;
    type = /* point create */ Attributes[DType(BCurrent)][/*@ *//D02B[DType(BCurrent)]]+
0];
    /* center */ (DCurrent[0]) = DATOBJ (Allocate (type));
    TRACELINES(8,"Create",
    /* center */ OBJDAT(DCurrent[0])) SEMICOLONFORTRACE

(*create_array [type])(
    /* center */ OBJDAT(DCurrent[0]),
DATREAL(100.0),
DATREAL(90.0));
    FCurrent[3 /* radius */] = ((float) (int32)80) ;

    (_02B001_gauge_draw_dial(BCurrent));
    DCurrent[17 /* title_len */] = DATINT((int32)8) ;
    type = /* string create */ Attributes[DType(BCurrent)][/*@ *//D02B[DType(BCurrent)]]+
+4];
    /* title */ (DCurrent[4]) = DATOBJ (Allocate (type));
    TRACELINES(14,"Create",
    /* title */ OBJDAT(DCurrent[4])) SEMICOLONFORTRACE

(*create_array [type])(
    /* title */ OBJDAT(DCurrent[4]),
DATINT((int32)8));

```

```

        (CALL(t1,
        /* title */ OBJDAT(DCurrent[4]),
        (_00200h_string_set(t1,
DATOBJ(OnceStr ("*****ALTITUDE")),
DATINT((int32)1),
DATINT((int32)8)), "set", 15));

        (_02B002_gauge_display_title(BCurrent));
DCurrent[18 /* units_len */] = DATINT((int32)5) ;
type = /* string create */ Attributes[DType(BCurrent)][/*@ */D02B[DType(BCurrent)]
+5];
/* units */ (DCurrent[5]) = DATOBJ (Allocate (type));
TRACELINES(20, "Create",
/* units */ OBJDAT(DCurrent[5])) SEMICOLONFORTRACE

(*create_array [type])(
/* units */ OBJDAT(DCurrent[5]),
DATINT((int32)6));

        (CALL(t1,
        /* units */ OBJDAT(DCurrent[5]),
        (_00200h_string_set(t1,
DATOBJ(OnceStr ("*****100 FT")),
DATINT((int32)1),
DATINT((int32)6)), "set", 21));

        (_02B003_gauge_display_units(BCurrent));
FCurrent[7 /* start_val */] = (0.0) ;
FCurrent[9 /* end_val */] = (10.0) ;
FCurrent[8 /* start_pos */] = (0.0) ;
FCurrent[10 /* end_pos */] = (((int32)2 * 3.14159265358979323846)) ;
FCurrent[11 /* start_label */] = (0.0) ;
FCurrent[12 /* end_label */] = (9.0) ;
FCurrent[13 /* label_inc */] = (1.0) ;
FCurrent[14 /* hash_inc */] = (0.25) ;

        (_02B004_gauge_draw_hash(BCurrent));
DCurrent[20 /* box_label_len */] = DATINT((int32)3) ;
type = /* string create */ Attributes[DType(BCurrent)][/*@ */D02B[DType(BCurrent)]
+6];
/* box_label */ (DCurrent[6]) = DATOBJ (Allocate (type));
TRACELINES(38, "Create",
/* box_label */ OBJDAT(DCurrent[6])) SEMICOLONFORTRACE

(*create_array [type])(
/* box_label */ OBJDAT(DCurrent[6]),
DATINT((int32)4));

        (CALL(t1,
        /* box_label */ OBJDAT(DCurrent[6]),
        (_00200h_string_set(t1,
DATOBJ(OnceStr ("*****Feet")),
DATINT((int32)1),
DATINT((int32)4)), "set", 39));
FCurrent[16 /* value_power */] = (100.0) ;
DCurrent[19 /* value_len */] = DATINT((int32)5) ;

        (_02A001_altitude_g_set_value(BCurrent,
DATREAL(0.0)));

```

```

;
_CR_INVARIANT;
_C_INVARIANT;
GACRSET;
TRACEOUT;
OUTJMP2;
return;
rescue: ;
    TRACERES;
    SETRES12;
end_rescue: ;
    PROJMP12;
    VIOLAT12;
    ERRJMP12;
}/*;*/

/*% set_value */DATUM .
_02A001_altitude_g_set_value (BCurrent,Local00N_value)
OBJPTR BCurrent;
DATUM Local00N_value; /* B2 */
{
    /*& redefine gauge set_value */
    static int routine=122;
    GACDEC1
    TRACEDEC
    _C_INVDEC
    int16 type;
    GACIN1;
    TRACEIN;
    SETJMP2;
    _C_INVARIANT;
    _C_SETINV1;

    (_02B007_gauge_set_needle(BCurrent,(Local00N_value)));

    (_02B008_gauge_set_box(BCurrent,(Local00N_value)));
;
    _C_SETINV2;
    _C_INVARIANT;
    GACRSET;
    TRACEOUT;
    OUTJMP2;
    return;
rescue: ;
    TRACERES;
    SETRES12;
end_rescue: ;
    PROJMP12;
    VIOLAT12;
    ERRJMP12;
}/*;*/

/*% keep02Ap */
void
keep02A(BCurrent)
OBJPTR BCurrent;
{
#ifdef KEEP
    static int routine = -4;
    DATUM *DCurrent;

```

```

float *FCurrent;
OBJPTR t1;

in_keep++;
DCurrent=Access (BCurrent)+/*@ *//D02A[DType(BCurrent)];
FCurrent = (float *)DCurrent;
(*keep_array [DT[161]])(BCurrent);
in_keep--;
#endif
}/*;*/

/*% Dive */
void
Dive02A (class_index, attr_off, feat_off)
int16 class_index, attr_off, feat_off;
{
    D02A[class_index] = attr_off;
    DD02A[class_index] = feat_off;
    Dive02B (class_index, attr_off + 0, feat_off + 0);
    {
        void Deal02A ();
        Deal02A (class_index, feat_off);
    }
}

/*% Deal */
void
Deal02A (class_index, feat_off)
int16 class_index, feat_off;
{
    InitRoutine (class_index, 122 + feat_off, _02A001_altitude_g_set_value, "set_value")
;
}

void
Init02A ()
{
    int16 DT160;
    DT160 = DT [160];
    Class_names[DT160] = "altitude_g";
    InitKeep(DT160, keep02A);
    Object_size[DT160] = 40;
    Interf_size[DT160] = 123;
    Routines[DT160] = (ROUT_PTR *) malloc(sizeof(ROUT_PTR) * 123);
    AllocRout_names (DT160 ,123);
    Attributes [DT160] = (int16 *) malloc (sizeof (int16) * 40);
    AllocAttr_names (DT160, 40);
    InitCreate (DT160, _02A000_altitude_g_create);
    Dive02A (DT160, 0, 0);
}

```