Investigation into Standardising the Graphical and Operator Input Device Modules for Tactical Command and Control Man-Machine Interfaces

Prepared by : Caireen E. Alston
Masters Student
Department of Electrical Engineering
University of Cape Town

Prepared for : Professor Mike R. Inggs
Department of Electrical Engineering
University of Cape Town

Mr Albert H. Broeksma
Development Department Manager
UEC Projects (Pty) Ltd
Cape Town

30 June 1994

Thesis prepared in Partial Fulfillment of the Requirements for the Degree of Master in Engineering.

ACKNOWLEDGEMENTS

I would like to thank my supervisors, Professor Mike Inggs, for his patience while I settled on a suitable topic, and Mr Albert Broeksma, for allowing me the opportunity to pursue a thesis topic related to my full-time work.

I would also like to thank all the team members at UEC Projects involved in the projects, for their help and guidance with the software packages and systems used.

My thanks also to Mr Chris Alston for proof-reading the final document.

SYNOPSIS

The operating environment of a Tactical Command and Control system is a highly tense one in which the operator needs to perform certain complex tasks with minimum confusion, and be able to obtain an instant response from the system. Since many of the systems designed for these types of environments are similar in nature with regard to the user-interface, a need has arisen to try and standardise certain elements of the systems.

This report looks specifically at standardising certain graphical display element and operator input device interfaces. It investigates the problem from a systems design level, identifying the elements required and their associated functions, discussing the results of work already undertaken in this field, and making recommendations on the use of the elements.

The main objective to standardising the Man-Machine Interface (MMI) design elements is to make the code easily transferable between different hardware platforms. To transfer the code, one would ideally like to change only the interface code to the new platform, in particular, the interface to a different set of operator input devices and a different type of graphics card.

Various topics related to the standardisation process are discussed, including a description of MMI design, some definitions of tactical command and control environment subjects, and a look at code reusability, rapid prototyping of systems, and object-oriented design.

The first major requirement of the standardisation process is the design of the graphical MMI display elements that are identified as being common to the various systems. This design includes the definition of co-ordinate systems, and an investigation of the use of colour to encode different meanings for groups or types of display elements, and the colours used for the different system modes of operation. The graphical MMI elements identified as being common to the various systems under investigation include: soft-buttons, totes, message bars, Plan Position Indicators (PPIs) and information boxes.

The second major requirement of the standardisation process is the identification of the operator input devices and the design of their interfaces. The input devices identified as being

common to the various systems under investigation include: a keyboard, softkeys, a joystick, and a touchpanel. Related to the design of the input device interfaces, is the design of input device event queues that handle all the inputs from the various devices, and convert these inputs into system events that can be handled by the system state-machine. Configuration files are used to enable the code to run on various hardware platforms that have either different types of input devices, or a different method of interfacing to the devices. Various hardware configurations and interfaces have been used to date. One of these configurations is a PC prototyping environment with a mouse and an IBM keyboard as input devices, using a PC serial port and the keyboard input. Another is a Multibus II (MBII) environment with an alpha-numeric STORM keyboard, STORM keypad softkeys, a joystick and a touchpanel as input devices, using serial input ports and MBII message passing.

The system operation as a whole is based on a scheduler, input device interrupts, and state-machines. The two main system state-machines are the logic-and-control state-machine, which interfaces with the system sensors, and the user-interface state-machine, which is discussed in this report. The user-interface state-machine uses displayed menus as the state-machine states, and decodes operator inputs into events for the state-machine, each of which has its own action function.

To illustrate the success of the standardisation design process, examples are given of systems already designed, and of work in progress. Photographs are used to illustrate the display screens and operator input devices used, and to give an indication of screen layouts and colour codings used.

From system design work already undertaken, the transportability of the software between different hardware platforms and system configurations has been successfully performed a number of times. This has made possible the use of low-cost PC-platform prototyping to speed up the design process, and has allowed frequent meetings with the users to obtain their comments during this design process. Recommendations arising from this work include the investigation of using other programming languages and packages that may speed up the design process even further. It is also recommended that the systems designed be observed under actual operating conditions.

ABBREVIATIONS AND ACRONYMS

ASCII	American Standard Code for Information Interchange.
Bit	Boolean piece of information, either 0 or 1.
Byte	Made up of 8 bits.
C^3	Command, Control and Communications.
COM Port	Personal Computer Communications Port.
COM1	PC Communications Port 1.
COM2	PC Communications Port 2.
DA	Designation Assembly.
GUI	Graphical User-Interface.
HCI	Human-Computer Interaction.
IBM	International Business Machines.
MBII	Multibus II.
MMI	Man-Machine Interface.
NDC	Normalised Display Co-ordinates.
PC	Personal Computer.
PCB	Printed Circuit Board.
Pixel	A point of display on a computer screen.
PPI	Plan-Position Indicator.
RGI	Raster Graphics Interface.
SKINT	STORM Keypad Interface.
SRCU	System Resources Control Unit.
STORM	Brandname of final system keyboard and keypads used.
TIGA	Texas Instruments Graphics Adapter.
UI	User-Interface.
Word	Made up of 2 bytes.
2-d	Two-dimensional.

KEYWORDS

Tactical Command and Control, Man-Machine Interface, User-Interface, Standardisation, Prototyping, Reusability.

TABLE OF CONTENTS

Page No

TABLE OF FIGURES

Page No

1. INTRODUCTION

1.1. Background

The operating environment of a Tactical Command and Control system is a highly tense one in which the operator needs to perform certain complex tasks with minimum confusion, and be able to obtain an instant response from the system. In order to design effective Command and Control systems, one has to look at the specific environment in which they have to operate, taking into account the functional requirements of the systems, the operators themselves, and the types of systems they are accustomed to using.

Since many of the systems designed for these types of environments are similar in nature with regard to the user-interface (UI), a need has arisen to try and standardise certain elements of the systems. This standardisation affects the system as a whole, from the actual console housing, to the hardware plug-in printed circuit boards (PCBs), to the types of operator input devices and man-machine interface (MMI) display elements. Standardisation of system equipment and layout is also recommended in MIL-STD-1472D on Human Engineering Design Criteria for Military Systems, Equipment and Facilities [27]. This standardisation refers to the common functions of all the equipment [27].

1.2. Scope

This report looks specifically at standardising certain graphical display element interfaces, and the operator input device interfaces. The report will look at the problem from a systems design level, identifying the elements required and their associated functions, discussing the results of work already done in this field, and making recommendations on the effective use of the elements. The systems design approach involves a study of the problem, and a definition of the requirements in pseudo-code format, as would be set out in a preliminary design document according

to DOD-STD 2167A [1]. No actual code is listed in the body of the report, but sample code from work done using the recommended standard features and functions is given in the appendices.

By standardising the MMI design elements, only the interfaces to the different hardware platforms have to be changed when the software is transferred between different platforms. These changes are specifically with respect to the graphics interface, and the interfaces to the various operator input devices.

1.3. Overview

The report starts by giving some background to the project, and discussing some related topics. Included in this background section is a description of **MMI design**, along with definitions of some of the terms used. The section then gives a brief description of a typical **tactical command and control operating environment** for which this investigation is intended. Three topics related to the standardisation process are discussed. The first topic covers the concepts and advantages of software **reusability** and maintainability as related to standardising software and software interfaces. The second topic is that of **rapid prototyping** of MMI systems, which is a useful tool for obtaining user feedback during the design stages of good user-interfaces. The third topic briefly dicusses **object-oriented design**.

Before describing the standard interfaces designed, an explanation is given of the different **hardware configurations** and platforms used, along with a description of the various **interfaces** required between the system and the graphics card, and between the system and the operator input devices.

When designing a graphical user-interface, thought has to be given as to how the different display information databases relate to the actual display routines in terms of where elements are to be displayed on the screen. This gives rise to the need for **co-ordinate system** interface definitions for the interfaces between database positional information, and display routine positional information. The choice of co-ordinate

systems is discussed.

The first major requirement of the standardisation process involves the design of **graphical MMI elements** that are identified as being common to various systems. These elements include soft-buttons, totes, message bars, plan-position indicators (PPIs) and information boxes. A description of these common MMI elements is given, as is a list of their required attributes and functions.

The **use of colour** plays a major role in the design of user-friendly user-interfaces. Different colours are used, for example, to group similar operations together, and to indicate the current system state and available operations to the operator. A discussion of the use of colour coding in MMI design is given, along with the choices of standard colours and their meanings for the systems under investigation.

The systems under investigation all have various **modes of operation**, such as operational and maintenance modes, in which such things as the rules of colour may differ slightly. These modes of operation are briefly discussed, with special emphasis on the different colour coding used.

The second major requirement of this investigation is the standardisation of the **operator input device interfaces** that have been identified as being common to the various systems. These input devices include a keyboard, softkeys, a joystick, and a touchpanel. A description of these common operator input devices is given, as is a list of their required attributes and functions.

Part of the operator input device element design is the design of **input device event queues** to handle all the inputs from the various devices, and convert these inputs into system events that can be handled by the system state-machine. The required event queues are identified, and the functions needed to match the inputs to the display elements are discussed.

In order to design a system that can run on more than one hardware platform without

it being necessary to re-compile the code, use is made of **configuration files**. These files are read by the program at startup to indicate to the system what platform and combination of devices is being used, so it knows how to handle the graphics and operator input. A discussion of the types of configurations used is given, along with sample configuration files.

The whole user-interface is controlled using **state-machine drivers**. Each state of these state-machines is the current displayed information. The events of the state-machine are as a result of operator input device events. The actions of the state-machine result in system operations being performed, and the displayed information changing to the next state. A brief description is given of the state-machine driver used for the systems under investigation, along with a template for the driver.

To round off the investigation into the standardisation of various MMI elements and input device interfaces, a **discussion** is given **of work already done** using the standards, along with a few **sample display screens** and code listings.

Finally, **conclusions** on the success of the design and interface standardisation are drawn, and **recommendations** on further improvements and work are given.

1.4. Limitations

The work described in this report was done as part of various project teams, working on several different MMI system projects. The author's contribution to the projects included the following: the design and layout of the MMIs, the design of PC prototype MMI systems, the design of the user-interface state-machine driver, the design and coding of the drivers for, and interfaces to operator input devices, and all the logical functionality of user interaction with display elements. Other team members were responsible for the actual specifications of co-ordinate systems, the coding of graphics drivers and interfaces for high level graphics systems, and for all the interface code providing interaction with the system sensors. Thus, as the topic of MMI design is complex, only certain aspects can be covered in a report like this.

2. RELATED TOPICS

2.1. Man-Machine Interface Design

A good user-interface is vital to any system, as it is the first point of contact a customer has with the system. An unfriendly user-interface will influence the customer's opinion of the rest of an otherwise good system. When designing user-interfaces, one must bear in mind the needs and abilities of the individual users [2].

User-interface software design can be complex, as it must monitor and control many devices, each of which is sending streams of input events to the system [3]. Stringent requirements are usually placed on system performance requirements such that the user perceives no lag between his actions and the system's response [3].

Some of the terms used in the design process can be used interchangeably, as will be done in this report. These include the terms Man-Machine Interface and User-Interface, and the terms operator and user.

When designing user-interfaces, thought has to be given to the type of interface as related to the types of users. User-interfaces for systems such as software packages and terminals are usually designed with on-line help facilities available at the push of a button, or the logical flow of operation indicates to the user the next function to perform. This reduces the need for intensive user training, as most users will easily be able to find their way around the system.

For systems that are to operate in a Command and Control environment, however, the user-interface is designed differently, in that no on-line help facilities are available. Because of the high level of tension present during operation, the system is designed to help the user by offering him only the choices available at a particular time. As all operators undergo intensive operator training courses on all the systems, help facilities are not really a requirement.

Consistency of the user-interface design is an important factor in its useability within an environment [4, 22]. If operators are required to use multiple interfaces within the same operating environment, consistent design aids their task considerably, leading to a higher performance level with fewer operator errors [4]. Users can become frustrated if systems do not behave understandably and logically, and consistency in the MMI design helps to avoid this frustration [5].

Ergonomics and common sense also play a part in user-interface design. This is especially true when deciding on the position of graphical elements and frequently used operations on the display in relation to the position of the operator and the operator input devices.

2.2. Tactical Command and Control Operating Environments

A Tactical Command and Control environment is one in which a central operator-controlled console is used by an operator to monitor and control an entire system. It is a real-time operating environment [6] that is extremely tense. The operator must constantly be kept informed of the current system state in as non-confusing a manner as possible in order for him to make the best decisions on what operations need to be performed.

Freiden [7] gives various definitions related to command, control and communications (C^3) systems, defining C^3 as "The knowledgeable exercise of authority in accomplishing military objectives and goals". The following definitions as given by Freiden [7] are relevant to the tactical command and control environment under discussion:

1) **Command** is the functional exercise of authority, based upon knowledge, to attain an objective or goal.
2) **Control** is the process of verifying and correcting activity such that the objective or goal of command is accomplished.
3) **Tactical** refers to narrowly defined methods of accomplishing objectives en route to a strategic goal. Normally a number of tactics make up a single

strategy.

4) **Operator Displays** provide sensor data and enable the operator to enter, readout, and make use of computer data via graphics and alphanumeric displays.

The heart of a good Tactical Command and Control system is a good MMI or user-interface. The task of the MMI is to keep the operator informed of the state of the various system elements (sensors, weapons, etc), and to provide him with adequate operations to control the operation of these system elements. The operators have to be able to make quick decisions under stressful circumstances based primarily on the information available to them on the MMI. Thus, this information has to be of such a nature that it is easily interpreted by the operators.

Some points given by Huckle [8] to be considered when designing MMIs for command and control applications include:

1) Allowing user participation in the design to make use of their knowledge and experience.
2) Symbols and colours should be used in such a way as to make the user-interface readable.
3) The amount of information presented to the user at any one time should be kept to a minimum.
4) A special interface for operator training may be appropriate.

2.3. Software Reusability

Reusability relates to the ease with which software can be modified to reflect changes in system design specifications. The whole drive behind the requirement for standardising the design of MMI elements and user input devices within the described command and control operating environment is so that the design effort is not wasted on each new project, and the ideas and code can be reused on subsequent projects. This saves both time and money, as new projects can use previous work and be completed sooner. The goals of reuse as given by Tracz [9], are "to improve

productivity, reliability, schedule, and reduce maintenance costs by exploiting what has been learned and used for similar programs in the past" .

As described by Agresti and McGarry [10], and by Sommerville [2], designing reusable software has the following benefits:

1) Productivity is increased [10], and development time decreased [2] as the designer can use existing components.
2) Reliability is better guaranteed, as the components reused have been proven in previous systems [10,2].
3) Overall risk is reduced by reusing components that have already been developed and tested [2].
4) Consistency of design is encouraged by reusing the same components in many places, reducing the need for redesigning new untested components [10]. This means that organisational standards can be incorporated in the reusable components [2]. For example, if a number of applications present users with menus, reusable menu components mean that all applications can present the same menu formats to the users [2]. This is the most important benefit that the command and control standardisation is trying to achieve.
5) Manageability of a project is made easier by using known components whose behaviour and reliability is already understood [10].
6) Reuse helps with standardisation, as the components are already available at the time of writing specifications [10].

In effect, the concept of reuse leads to standardisation, as code that is frequently reused becomes standard to the systems.

Related to the design of reusable code is the subject of maintainability. Designing code for reuseability within similar systems also means that the code should be more maintainable. Once the elements and their associated functions and attributes have been identified, standardised and properly documented, these elements can be coded into the language required by the current system. Samples of elements that have already been coded can be kept in a library for reuse by other designers with the same requirements. Systems designed using the standard elements are easily maintainable,

as the code has usually been used and tested in previous systems where problems and faults have been ironed out and documented.

An aid to writing reusable software is the use of object-oriented design [9]. Each reusable element of the design can be written as an object to be initialised with different parameters depending on the current application it is being used for.

2.4. Rapid Prototyping

"Prototyping is the process of building a working model of a system or part of a system" [11]. Prototyping is used in engineering to obtain an early version of a product or system, and is critical in the design and development of user interfaces that are of high technical quality [12]. It is a very valuable technique for helping to establish system requirements [2, 25]. The objective of a prototype is to clarify the characteristics and operation of a product or system by constructing a version that can be exercised. Prototyping aids the design process when user's requirements for the system are not well understood. The prototype attempts to present the user with a relatively realistic view of the system as it will eventually appear. Users are thus able to relate what they see in the form of a prototype directly to their requirements [13].

The advantages and benefits of using prototypes during system design include [12,13]:

1) Obtaining valuable user feedback early in the design process [12,14], which leads to improved functional requirements [13,14].
2) Operator performance data can be collected within the context of the specific application [12], which leads to improved inter-action requirements [13].
3) The prototype provides a common point for user-interface designer-developer communication so that problems can be identified early in the development process [12], which leads to the easier evolution of requirements [13].
4) Quick modification for iterative design and test is encouraged, providing a potent opportunity for an improved user interface.

Figure 1 illustrates prototyping in the software process [2], where iteratively developing and evaluating a prototype leads to the system specifications and design.

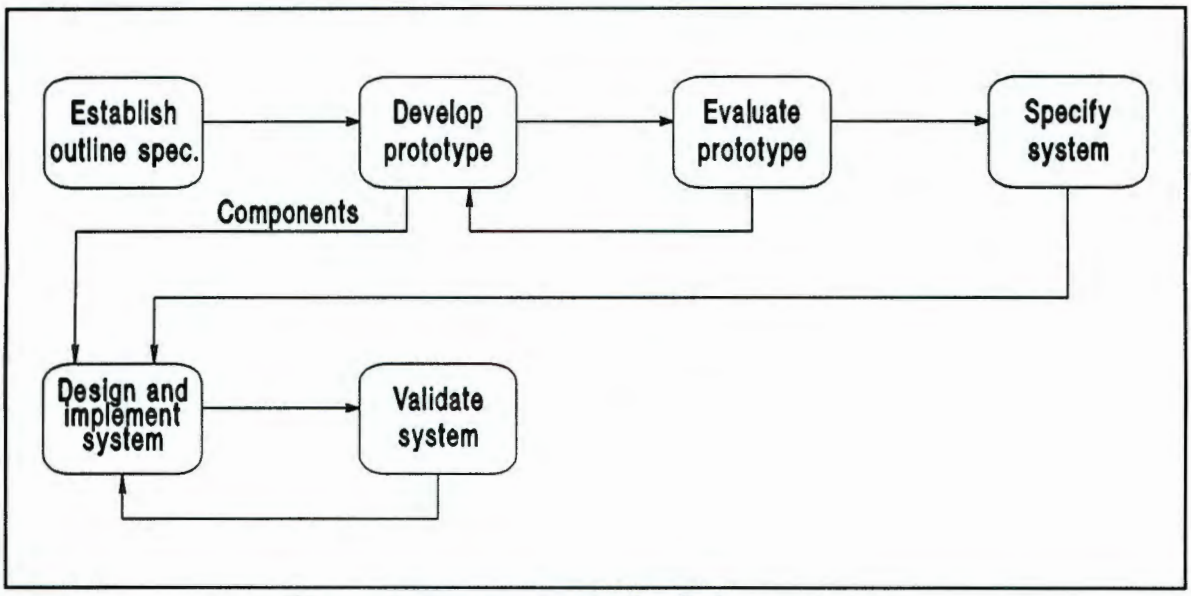

Figure 1: Prototyping in the Software Process

When designing a system that emphasises the MMI and how the operator interfaces with the system, there is a need to have frequent meetings with the users during the system design stages to gain feedback from them on what they feel about the design. These meetings give rise to the need for a rapid prototyping capability which allows the programmer to change the MMI elements in accordance with the user's comments. A prototype user-interface is much more effective than documentation and drawings for defining the final system specifications. Development through prototyping is an iterative process, as with each prototype new features are demonstrated and comments obtained from the user. Changes made as a result of previous comments are reviewed, and new changes are made as required. If the final iteration of the prototype is functionally complete and technically sound, it may actually become the operational system [15, 22].

As described earlier, the standardising of the MMI elements should only require the rewriting of the interfaces to the graphics card and to the various operator input devices. This makes rapid prototyping relatively easy. It is usually easier to create a prototype in a PC environment instead of on the final operating hardware platform.

This serves two purposes. Firstly, the MMI design can be done in parallel to the design of the hardware platform, and secondly, the prototype can be taken to the user to gain their input on any changes they would like to see, instead of them having to wait for the whole system to be operational. Prototyping also needs to be done early in the project so as to obtain user feedback before the requirements specification document is finalised. Based on user feedback, the requirements specification can be revised for the final system design.

Prototype systems can also be used for training, as certain aspects of the MMI can be shown with a prototype system while the complete system is still being completed or installed.

2.5. Object-Oriented Design

Object-oriented design refers to the way in which software is organised as a collection of discrete objects, each of which incorporates its own data structures, and has its own behaviour [26]. An object-oriented design approach is a useful tool for user-interface design, as each display element and input device can be represented and coded as a separate object, to be initialised with different parameters depending on the current application it is being used for.

Object-oriented design provides many advantages for the design process. Once objects have been properly defined, tested and documented, they can be stored in a library for use in future projects. Thus, objects can be reused from the library as building blocks in new projects [21,23]. This reuse results in the system development process being speeded up, allowing rapid prototyping to take place [21].

3. HARDWARE CONFIGURATIONS AND INTERFACES

The main drive behind the standardisation requirement is so that code written for any one system is reusable with other systems that may reside on different hardware platforms. The transfer capability of the code also enables easy design and prototyping on a PC.

3.1. Hardware Configurations

Two main hardware configurations are used, namely a PC development and prototyping configuration, and the main system configuration residing on a Multibus II (MBII) platform.

The PC development and prototyping configuration consists of a 14-inch monitor using VGA graphics, and operator input devices of an IBM keyboard plugged into the PC keyboard input, and a PC mouse plugged into the PC serial port. This configuration is shown in Figure 2.

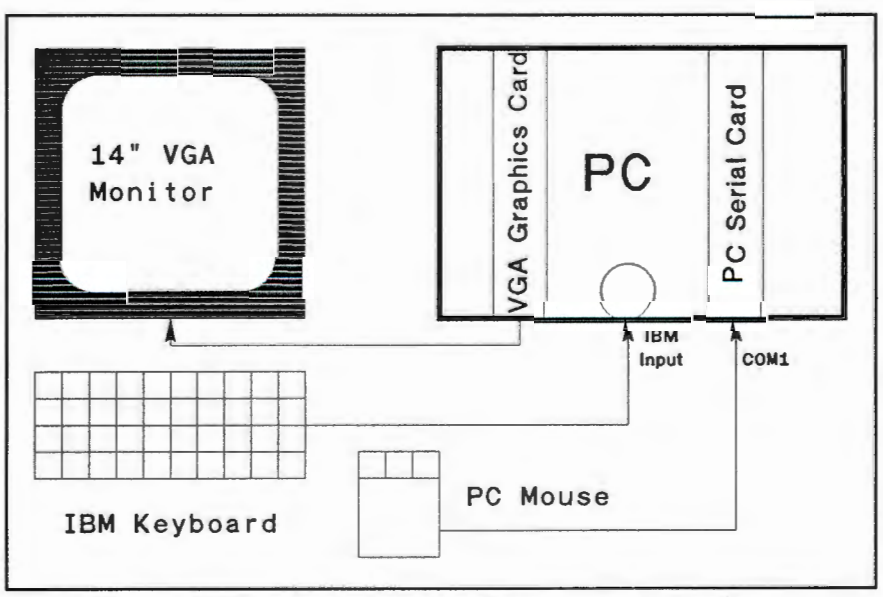

Figure 2: PC development and prototyping hardware configuration.

The MBII final system platform configuration consists of a 19-inch monitor using high level RGI TIGA graphics, and various operator input devices. These input devices interface to the system via serial ports on the MBII cards. The input devices consist of a joystick and a touchpanel, which interface directly to the serial inputs, and a STORM keyboard and set of STORM keypad softkeys, which interface to the serial inputs via a SKINT PCB. The MBII hardware configuration is shown in Figure 3.

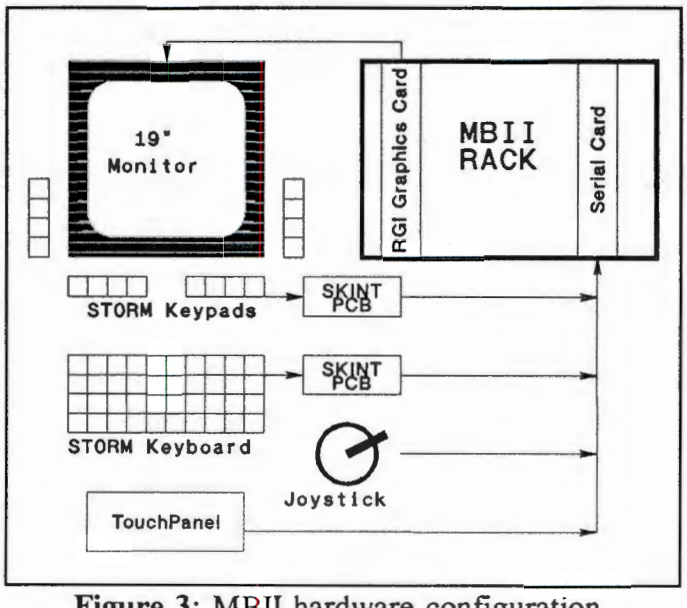

Figure 3: MBII hardware configuration.

3.2. Code Layers

The system comprises a layer of standard code modules that interface to the hardware via interface code module layers. In the case of the graphics, the main program places all display items in a display server, which interfaces via the display driver to the graphics card. In the case of the operator input devices, each device interfaces to the program code using its own device driver, which places decoded input events into event queues for handling by the program. When transferring code between hardware platforms, only the interface code should change, the rest of the code must be system independent. These system layers are shown in Figure 4.

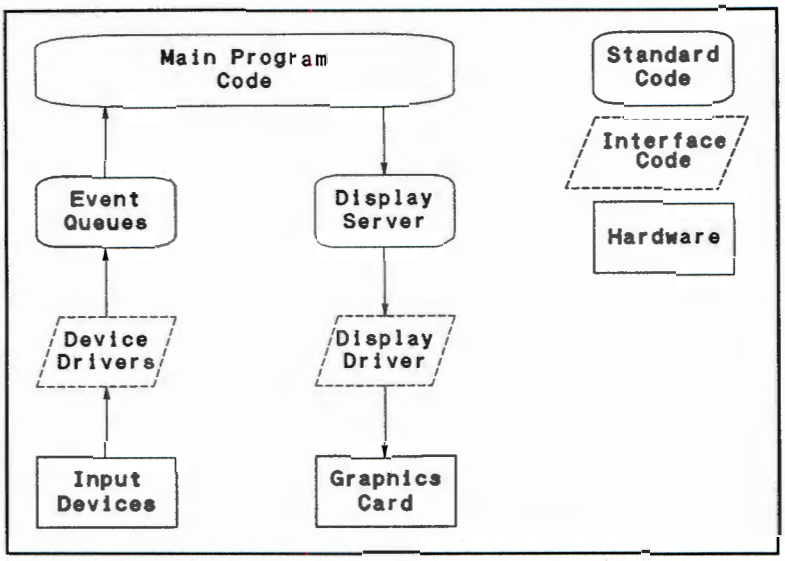

Figure 4: Graphical representation of the code layers and interfaces.

3.3. Interfaces

There are two main sets of interface code required, namely the input device interface code, and the graphics driver interface code.

The input device interface code must interface to the various operator input devices used for the different hardware platforms. In the PC configuration, interfaces are required to the PC serial ports for mouse inputs, and to the IBM keyboard input. In the MBII configuration, interfaces are required to the serial ports of the MBII cards into which the input devices are plugged. Figure 5 shows the interfaces between the system and the operator input devices.

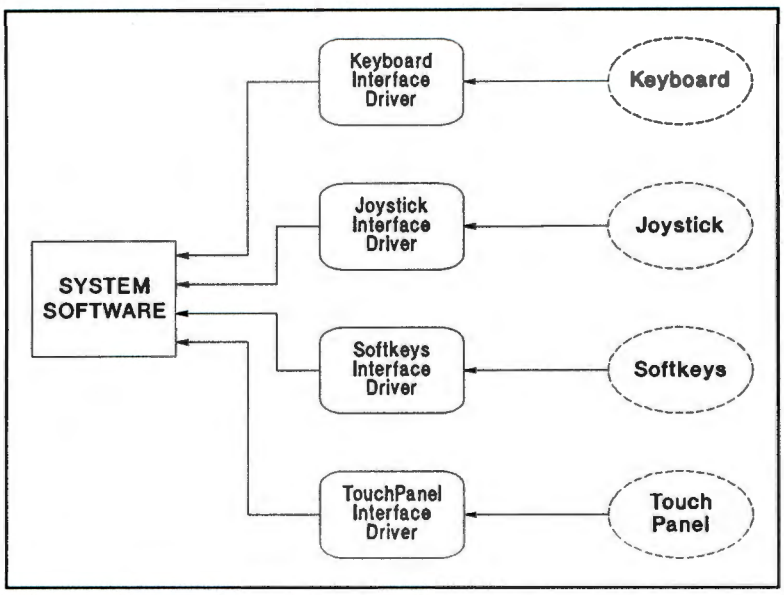

Figure 5: Interfaces between the system and the operator input devices.

In the PC configuration, the graphical interface is to a VGA graphics card that displays the MMI on a 14-inch monitor. The MBII final system configuration uses an RGI Tiga graphics card that displays the MMI on a 19-inch monitor. Figure 6 shows the interface between the system and the graphics card.

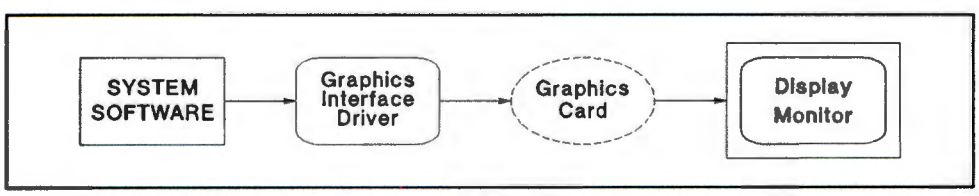

Figure 6: Interface between the system and the graphics card.

4. COMMON GRAPHICAL MMI ELEMENTS

As a result of studying existing command and control systems in the operating environment, and after discussions with users and operators, certain common MMI elements, system operations and future requirements became evident.

The design of the standard MMI objects that were identified starts with the definition of all the functions that are required by the objects. These functions then give rise to the structures and variables required by the objects, and header files can be designed and coded to provide the correct information.

Graphical MMI elements identified as being common to all the systems in the operating environment include soft-buttons, totes, message bars, PPIs, PPI elements, and information boxes. Most of the MMI objects have both graphical and logical operations associated with them. For systems written in languages like C++ that directly support object-oriented programming, the split between graphical and logical operations is easily achieved using the inheritance property of classes. For systems written in languages such as normal C, the split is less easily achieved, and both attributes are usually combined into one file.

The following sections deal with each of the standard MMI objects identified. A short description of each object is given, followed by a description of all the functions that must be provided by the objects. As the sections of this document dealing with the graphical and input device elements are in the form of a requirements specification, the required functions are listed in pseudo-code, along with descriptions of their operation. Sample code listings of some of the actual system module header files are given in Appendix B and Appendix D.

In order to design the graphical MMI elements, co-ordinate systems have first to be defined to handle the interfaces between the databases and the display. The co-ordinate systems are described before the graphical elements and their functions are defined.

4.1. Co-ordinate Systems

As part of the standardisation process, a common co-ordinate system had to be defined that is independent of the graphics platform, screen size, etc. This is so that the functions of the standard MMI elements can be designed and coded independent of the graphics platform of the final systems. Thus, only a new graphics driver needs to be coded for a new platform instead of the whole display database needing recoding.

A co-ordinate system was designed in which all positions and dimensions of MMI elements are specified in Normalised Display Co-ordinates (NDCs). These NDCs define the bottom left of any display as the origin, independent of the screen orientation or where the screen origin is [5]. An NDC point is made up of a horizontal (**h**) and a vertical (**v**) component. Any calls made by the MMI elements to the graphics driver routines pass NDCs to these routines. The graphics driver that is written for each particular graphics platform then has to manage the conversions from NDCs to pixels depending on that particular graphics system, the screen size used, and the screen orientation. The pixel origin, however, is fixed and is dependent on the screen orientation. A pixel point is made up of an **x**-component and a **y**-component. Figure 7 shows the different display sizes and orientations used by the various systems.

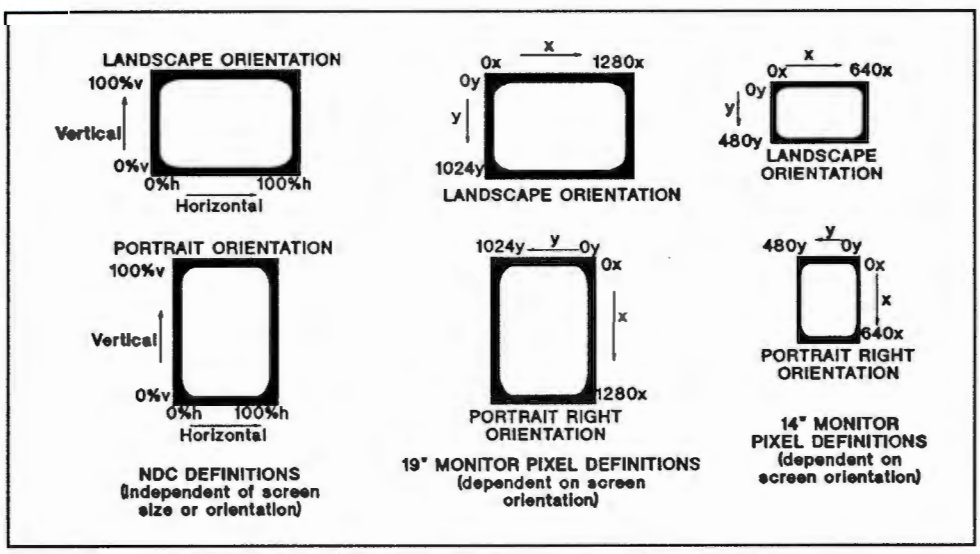

Figure 7: Different display sizes and orientations.

The operator input devices, such as the joystick and the touch panel, receive as their input, as a result of an operator action, a position in graphical pixels. This pixel position must then be converted by the input device drivers to NDCs so that the event can be processed by the code of the affected standard elements, and matched with element positions in the database.

4.2. Soft-Buttons

4.2.1. Description

Soft-buttons are used to perform operator system functions, providing various levels of menu functions for selection. Button selections are used as events for the system state-machine. The display positions of these buttons are required to match up to the softkey positions so that selection is possible via either the softkeys or the touchpanel.

The soft-buttons are display objects that have both logical and graphical attributes. The graphical attributes include such functions as the button's display position, the colour in which to display the button, and the button's label. The logical attributes include such functions as setting whether or not the button is available for selection, and the action function that must be called when a button is activated. This action function controls the state of the buttons.

From the study of display and functional requirements, it was noted that different types of buttons would be required. These include buttons that can be divided into a maximum of 4 parts, each part having its own state and label. The button is still activated as a whole, and only has one action function associated with the whole button. The purpose of the different parts of a button would be to act as an indication of system states associated with the function of that button. For example, a button in the main menu that is used to call up the 'Range' submenu to set the required range would be a two part

button. The top half of the button would be displayed in white to indicate that the function is available for selection, with the label "RANGE", and the bottom half of the button would have a label indicating the current range that is selected, and would be displayed in green to show that it is selected.

Figure 8 shows what displayed buttons typically look like, with the different parts, colours and labels. To make the button emulate a mechanical button push as closely as possible, the button has a graphical shadow around it. When a button is pushed, it is drawn in reverse video to emulate the button being depressed **in** and released **out**.

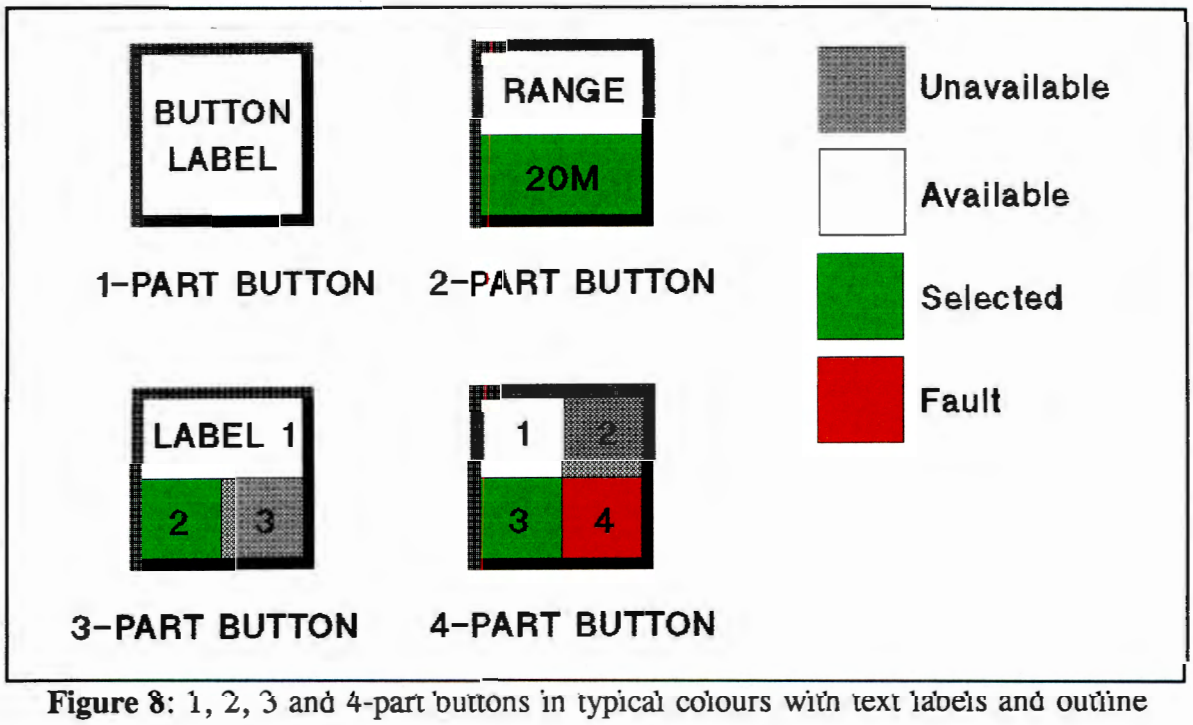

Figure 8: 1, 2, 3 and 4-part buttons in typical colours with text labels and outline shadows.

4.2.2. Functions to Initialise a Button

The following function is used to initialise the various types of buttons required. Once a button has been initialised, its displayed position does not change, even though it may be displayed or hidden as required.

i) BUTTON_Initialise()

This function is required to initialise a button with certain parameters. These parameters include the position at which the button is to be displayed, whether or not the button is active and available for selection, a pointer to the action function to be called on activation of the button, the number of parts the button is to have (1 to 4), the labels for each button part, and the state of each button part, which determines the background colour in which it is to be drawn.

4.2.3. Functions to Handle Logical Button Attributes

The following functions are used to perform logical button operations. It is noted that some of the graphical functions described in the next section, such as for the setting of button state and label, are also in part logical functions, as they are called by the system state-machine as well as by the graphics module, to determine present button states, and to set new states and labels.

i) BUTTON_Active()

This function is required to access the button's active state. The state is set to TRUE if the button is active and available for selection, and to FALSE if the button is inactive and unavailable for selection.

ii) BUTTON_Action()

This function is required to access the pointer to the button's action function. This action function is called each time the button is activated as a valid state-machine event.

iii) BUTTON_State()

This function is required to access the state of a specified button part which indicates the background colour in which the button must be displayed. A change in button state is usually as a result of a button event in the state-machine.

iv) BUTTON_Label()

This function is required to access the label of a specified button part. A change in label is usually as a result of a button event in the state-machine.

4.2.4. Functions to Handle Graphical Button Attributes

The following functions are used to perform graphical operations on the buttons, such as drawing and hiding the buttons.

i) BUTTON_Changed()

This function is required to indicate that some feature of the button has changed and that a redrawing of the button is required.

ii) BUTTON_Draw()

This function is required to draw the button according to its currently stored attributes.

iii) BUTTON_Hide()

This function is required to hide a button by erasing it from the display. The button's attributes are still stored for redrawing the button when required.

iv) BUTTON_Pushed()

This function is required to indicate that the button has been pushed, so that it can be redrawn in such a way as to simulate the depression of a mechanical button.

v) BUTTON_Inside()

This function is required to determine if a given NDC point position on the screen falls within the boundaries of the button's displayed position.

4.3. Totes

4.3.1. Description

A tote is a matrix-like box displayed on the screen to provide information on the states of various system elements in a tabular format. This information is held in a system scenario database. A tote may have access to more information than can be displayed at any one time, in which case scrolling functions are provided to scroll through all the available information.

Figure 9 shows typically what a displayed tote looks like, with the different toteline colour codes, typical headings and displayed information.

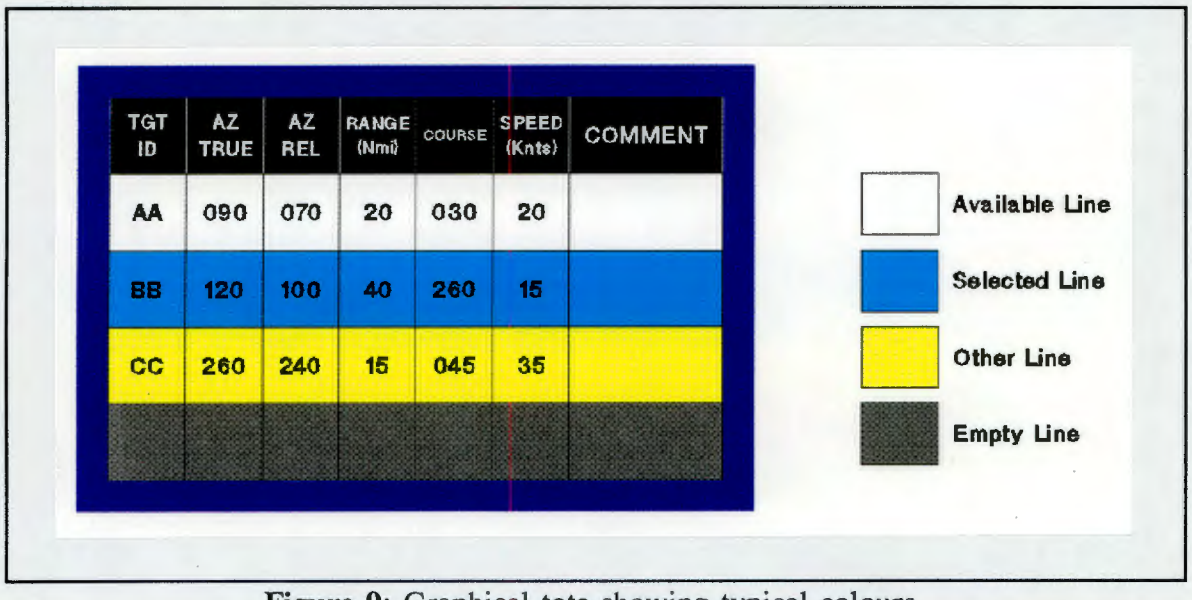

Figure 9: Graphical tote showing typical colours.

The tote is a display object that has both logical and graphical attributes. The graphical attributes include such functions as the tote's display position, the colour in which to display the different elements of the tote, and the textual information to be displayed in the different tote elements. The logical attributes include such functions as setting which target's information to display in which toteline, and which toteline is currently selected.

4.3.2. Functions to Initialise a Tote

The following function is used to initialise a tote according to the required attributes.

i) TOTE_Initialise()

This function is required to initialise the tote with its display position, the number of rows and columns, and any headings.

4.3.3. Functions to Handle Private Tote Attributes

The following functions are used to perform private actions on the tote as a result of external function calls or events.

i) TOTE_SelectEntity()

This function is required to set the state of the entity displayed in the specified toteline to **selected** or **deselected** in the scenario database.

ii) TOTE_AddEntity()

This function is required to add a new entity to the tote by either placing it in an empty toteline, or by replacing the least important entity currently displayed.

iii) TOTE_EntityAlreadyInTote()

This function is required to determine if a specified entity is already displayed in the tote.

4.3.4. Functions to Handle Logical Tote Attributes

The following functions are used to perform logical operations on the tote, such as scrolling.

i) TOTE_Scroll()

This function is required to handle the scrolling of information in the tote, both **up** and **down**. All the entities available for display are stored in a tote buffer, and are scrolled on and off the tote as required.

ii) TOTE_Empty()

This function is required to determine if the tote is empty, ie: if there are any entities currently displayed in the tote.

iii) TOTE_EntitiesInBuffer()

This function is required to determine the total number of entities currently in the tote buffer.

iv) TOTE_Execute()

This function is called by the scheduler each time the tote is scheduled to run. It allows for management of the tote elements in terms of entities that are no longer valid and need to be dropped from the buffer and the display, entities that have to be added to the buffer and displayed, entities that require selection or deselection, etc.

4.3.5. Functions to Handle Graphical Tote Attributes

The following functions are used to perform graphical operations on the tote, such as drawing and erasing it.

i) TOTE_Draw()

This function is required to draw the entire tote. The function uses the entity numbers of each toteline to extract the relevant information on that entity from the scenario database for display in that toteline. The state of the entity as obtained from the database indicates in which colour to draw the toteline.

ii) TOTE_Erase()

This function is required to erase the whole tote from the display if the need arises.

iii) TOTE_Inside()

This function is required to determine inside which toteline a specified point position on the screen falls.

4.4. Message Bars

4.4.1. Description

A message bar is a display object that presents a message to the operator inside a rectangular box in the desired colour associated with the type of message. Message bars are used to display prompt messages to aid the operator on what functions to perform next, error messages to indicate system faults or incorrect operator choices, and for general system state messages.

Figure 10 shows typically what a displayed message bar looks like, showing the different colours associated with the different types of messages.

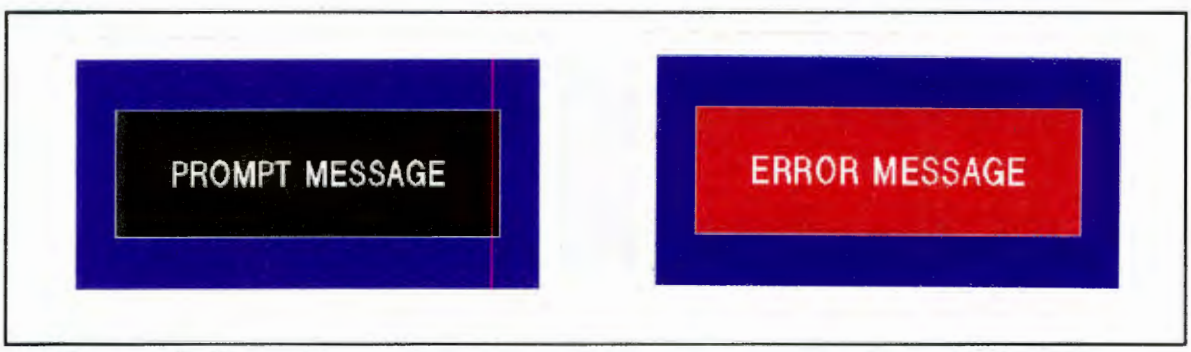

Figure 10: Typical prompt message and error message bars in required colours.

4.4.2. Functions to Initialise a Message Bar

The following function is used to initialise a message bar with the required attributes.

i) MESSAGEBAR_Initialise()

This function is required to initialise the message bar with its display position, initial message, and background and text colours.

4.4.3. Functions to Handle Logical Message Bar Attributes

The following functions are used to perform logical operations on a message bar, such as scrolling through past messages.

i) MESSAGEBAR_Message()

This function is required to set a new message to be displayed with the required text and background colours.

ii) MESSAGEBAR_PromptMessage()

This function is required to set a new prompt message to be displayed. The background and text colours are preset according to the prompt message colour rules.

iii) MESSAGEBAR_ErrorMessage()

This function is required to set a new error message to be displayed. The background and text colours are preset according to the error message colour rules.

iv) MESSAGEBAR_Scroll()

This function is required to handle the scrolling of the message bar if there are more messages in the message buffer.

4.4.4. Functions to Handle Graphical Message Bar Attributes

The following functions are required to perform graphical operations on the message bar, such as drawing and erasing it.

i) MESSAGEBAR_Draw()

This function is required to draw the message bar if any of its attributes has changed.

ii) MESSAGEBAR_Erase()

This function is required to make the message bar invisible by erasing it from the display.

iii) MESSAGEBAR_ChangePosition()

This function is required to change the position of the message bar during program execution.

4.5. Attributes Common to the Display Elements

As can be seen from the above sections, many of the functions required by these common MMI graphical elements are in turn common to all the elements. For instance, all the elements require functions to set their positions, draw and erase them, set the background drawing colour and the text colour, set the actual text to be written inside the elements, and a function to determine if a point is inside the element or not. Depending on the design language used for a particular system, for example C++, it may be possible to combine all these common functions into one class or object, such as a 'TextBox' class. The MMI elements can then be derived from this textbox class, and can inherit the functions it supplies.

4.6. Other Common Display Elements

It is noted that the scope of this investigation did not extend to the standardisation of screen elements other than those already described. That is, the investigation was concerned only with those screen elements that have distinct logical functions associated with them. Various other standard graphical elements are indicated here, but are not discussed in detail.

4.6.1. Plan-Position Indicator (PPI)

The PPI is a circular area, usually in the centre of the screen, used to display raw radar video as received from a search radar. The current positions of the various system sensors and other information is displayed along with this raw radar video. The PPI display is then used by the operators to analyze situations and act accordingly. The following section describes some of the information displayed along with the radar video in the PPI, and Figure 11 shows a typical PPI display.

4.6.2. PPI Elements

PPI elements are objects that are displayed within the PPI radius itself. Figure 11 shows a typical PPI display with the common elements displayed in their respective colours. Elements that have been identified as being common to the various systems' PPIs include:

i) Compass Rose Rings

A true compass rose is displayed in yellow or white on the outside circumference of the PPI, with labels every 10 degrees. A relative compass rose is displayed on the inside circumference of the PPI, and has a green arc on the starboard 180 degrees, and a red arc on the port 180 degrees, each with labels every 10 degrees.

ii) Strobe and Cursor

This consists of a line (strobe) radiating from the PPI centre to the PPI circumference indicating the cursor bearing, and a small circle (cursor) running along this vector indicating the cursor's current range.

iii) Sensor Vectors and Symbols

These are displayed inside the PPI and differ between the various systems depending on the sensors specific to those systems. The sensor vectors are lines radiating from the PPI centre to the range and bearing at which a sensor is currently looking. The sensor symbols are shapes, such as circles or triangles, that are drawn at the range and bearing at which the sensor is currently looking.

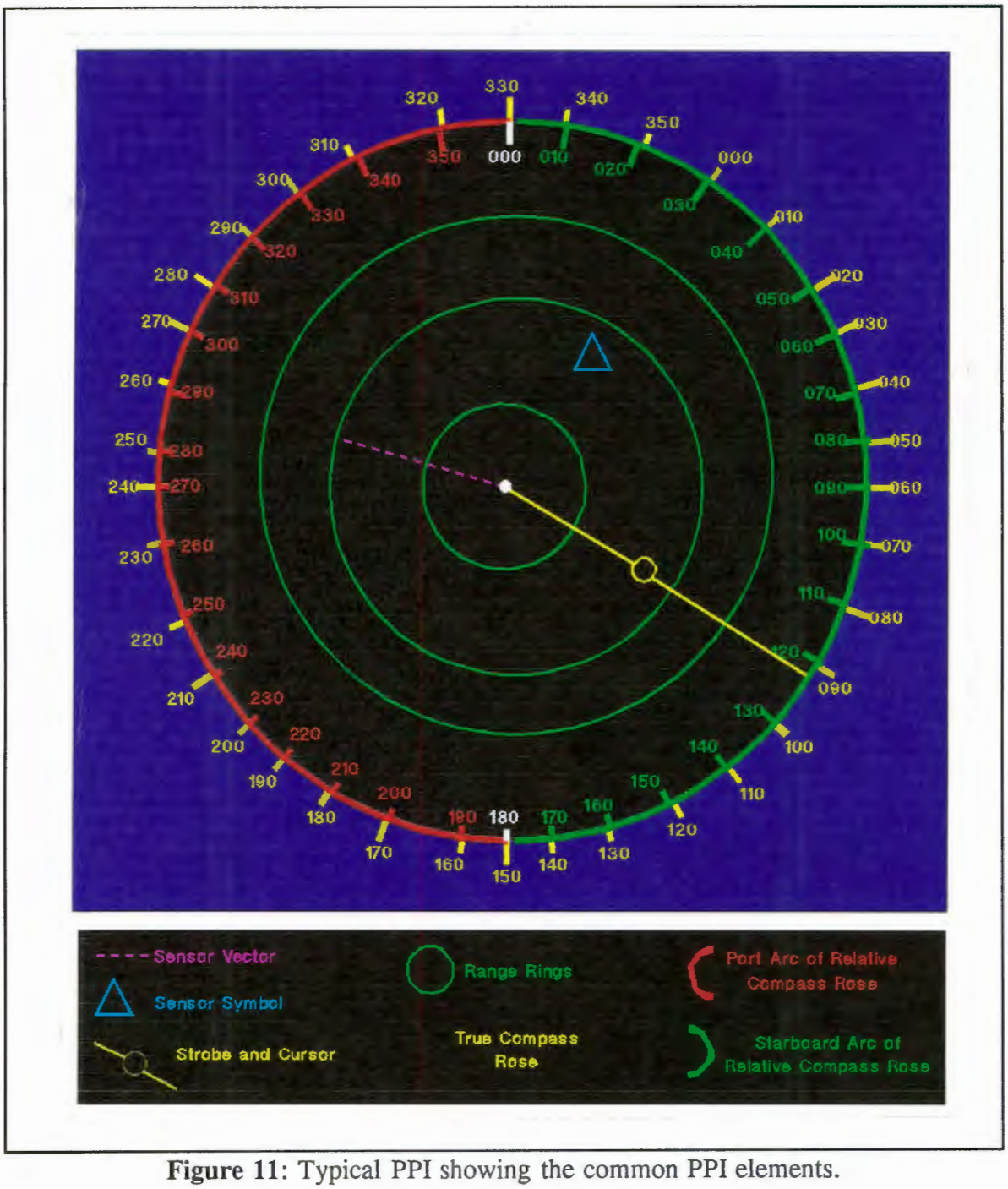

Figure 11: Typical PPI showing the common PPI elements.

4.6.3. Information Boxes

These are boxes that can be displayed wherever required, and that hold various bits of information required by the operator, but not necessarily vital to the system's operation. They can thus be displayed in empty corners of the screen, as they do not have to be central to the operator's field of vision. Information boxes identified as being common to most systems include:

i) Ship's Position Box

This is used to display the ship's current course and speed in the required units of measure of the specific system. A typical ship's position box is shown in Figure 12, displaying the information in the units for a particular system.

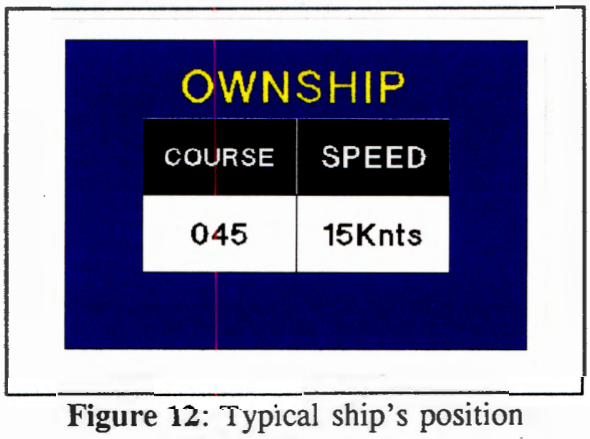

Figure 12: Typical ship's position information box.

ii) Cursor Position Box

This is used to display the cursor's range and bearing within the PPI dependent on the current range setting, in the required units of measure of the specific system. A typical cursor position box is shown in Figure 13, displaying the information in the units for a particular system.

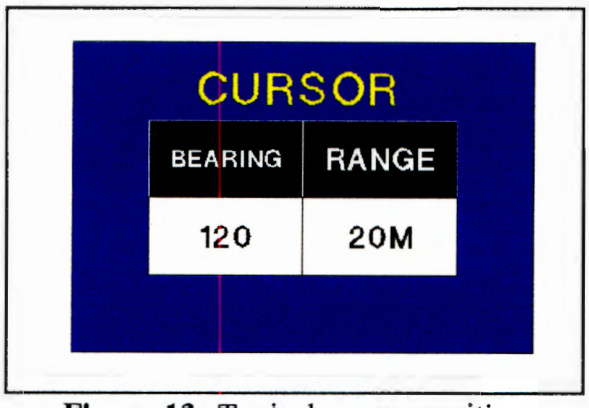

Figure 13: Typical cursor position information box.

4.7. Colour Coding

4.7.1. Rules for Colour Coding

Colour coding is an important means of conveying information to people. It is also an efficient means of information transfer [16], as a user can easily associate colours with states rather than having to read detailed labels or information. Thus, colour encoding of display objects and groups of similar information improves user performance.

In his article on rules of thumb for display colour coding, Rice [16] sets out 10 rules of thumb to aid user-interface designers in the correct use of colours. Schneidermann [17] and Sutcliffe [18] also give some guidelines on the use of colour in user-interfaces. Some of the important rules and guidelines include the following:

1) Colour coding can be used to: show status or significance [16,18], draw attention, order data, separate data, and to group or show similarities [18]. Such colour associations will ease the user's task [16].
2) The display background should be of a neutral colour so as not to detract the user's focus from the actual interface information [16].
3) The lighting of the environment in which the system operates is related

to the choice of display colours, and vice-versa [16].

4) The number of different colours used should be limited to prevent operator confusion [16,17].

5) Colours should be used consistently [17].

6) Colour coding should take into account the expectations of the operators in terms of using everyday colours everyone is familiar with [22].

Another interesting concept in the choice of display colours, as indicated in many of the references, is that a significant percentage of the male population (8% in the USA [16]) suffers from some form of colour blindness. Rice [16] suggests one solution is to use only certain colours that are visible to both colour-blind and normally sighted people. These colours include: blue, green (or white), red, cyan, and yellow-green. The effectiveness of colour choice obviously still depends on the type of colour blindness or colour perception present in an operator.

As part of the standardisation process for this project, colour coding of various MMI elements to mean different things to an operator is very important. At a glance, the operator should be able to know what state the system is currently in by the colours that the various button functions, tote lines and message bars are displayed in.

Examples of the use of colour in systems can be seen in the diagrams of Section 4 on the Common Graphical MMI Elements, and in the photographs shown in Section 8 on the Practical Implementation of the Standard Design Elements.

4.7.2. Soft-Button Colours

Common MMI element background colours as related specifically to soft-button system operations or functions that were identified include:

1) Grey, which denotes that a certain function or operation is currently unavailable for selection by the operator.
2) White, which denotes that a certain function or operation is currently available for selection by the operator.
3) Green, which denotes that a certain function or operation is currently selected or in progress, but may still be available for re-selection by the operator.
4) Red, which denotes that there is an error with either a system function, or with part of the system itself.

These colours were chosen on the basis of colour associations we encounter in everyday situations, such as green indicating **'OK'**, while red indicates a fault, and on the basis of colours commonly known to operators of older analogue systems. In the older systems, analogue switches were used for user input. These switches had back lighting to indicate certain element states, such as green for selected and white for available. A major drawback to the analogue systems is that the backlighting cannot indicate to the operator when an operation is unavailable. Thus operators have to learn the system, and from memory know what operations they can perform when. The analogue colours were also adopted in the colour standardisation, as they are logical and will aid users transferring to new systems.

4.7.3. Tote Colours

Common MMI element background colours as related to tote information and operations that were identified include:

1) Grey, which denotes that the line does not yet contain any information.
2) White, which denotes that a line of information is available for selection by the operator.
3) Cyan, which denotes that a line of information has been selected by the operator.
4) Yellow, which denotes one other possible state of a line.

Older systems make use of separate monochrome monitors to display tote information, so the choices of colours are based on common-sense colour knowledge, and on input from the users.

4.7.4. Message Bar Colours

Common MMI element background colours as related to message bars were identified as:

1) Grey, which denotes that no messages are currently to be displayed.
2) Black, which denotes a prompt message, or a message of general information or reporting.
3) Red, which denotes a message indicating a system error.

Older systems do not have any form of message interfacing with the operator, so the choices of colours are based on common-sense colour knowledge, and on input from the users.

4.8. System Modes of Operation

4.8.1. Operational Mode

This is the normal mode of system operation in which all the system functions are performed. The colour coding of the various MMI elements is as described previously, and sample operational mode screens are shown in Section 8 on the Practical Implementation of the Standard Design Elements.

4.8.2. Maintenance Mode

This mode of operation is entered by means of a certain keystroke combination and a password, thus enabling only authorised operators to access maintenance mode. From maintenance mode, the operator can return to operational mode with a single keystroke.

In maintenance mode, a series of sub-menus are provided allowing the operator to examine the state of various system elements. This mode serves as a helpful fault-finding tool, as by colour coding the various sub-menu buttons which relate to the system elements, the operator can easily identify the faulty element.

The common colours identified for use in the maintenance mode include:

1) Green, which denotes that the specific system element is functioning properly and is not faulty.
2) Red, which denotes that the specific system element is faulty and can be replaced.
3) Cyan, which denotes that the state of the specific system element cannot be determined, which may be an indication that it is faulty.

4.8.3. Setup Mode

This mode is only available for some systems, and allows the input of information related to certain system states that may otherwise not be available in the operational or maintenance modes. For example, if a particular sensor is not available, it may appear as faulty in the maintenance mode. An operator with rights to the setup mode would be able to enter this information into the system that the sensor is not currently available, and this would then be taken into account when reporting system states.

5. COMMON OPERATOR INPUT DEVICES

Operator input devices are used by the operators to interact with the system by allowing them to select various MMI operations and for the selection of certain MMI elements. To decide what input devices are required, and to design their operations, thought needs to be given to the types of operations required by the system.

Associated with the design of the operator input devices is the ergonomics involved as to where to position for instance the keyboard and the joystick in relation to each other and the MMI screen. When designing ergonomically, one must take into account the system operations that can be performed in relation to the devices needed for those operations. The position of the operator and the number of hands required at any one time to perform an operation must also be considered.

Many of the MMI functions can be performed by more than one of the input devices. This is one of the system requirements identified so that in the event of one of the input devices failing, the system can still be operated using the other devices to perform the same functions. As this is a degraded mode of operation, the system operations affected may not be as easily performed, but will nevertheless be available to the operator.

5.1. System Configurations

Each operator input device has its own standard functions that it requires to operate. These functions are independent of the hardware platform in which the system is housed, so when redesigning for a different system, only the interface to the hardware platform needs to be changed.

Figure 14 shows a typical PC development and prototyping configuration, with a mouse and an IBM keyboard as the only operator input devices. This configuration usually uses a 14" monitor with VGA graphics. Figure 15 shows a typical final system operator console with STORM keyboard, STORM keypad softkeys, rollerball

and touchpanel as the operator input devices. The final system consoles usually use 19" monitors and high-level graphics.

Figure 14: Typical PC prototype configuration with mouse and IBM keyboard as operator input devices.

Figure 15: Typical final system operator console with keyboard, softkeys, rollerball and touchpanel as operator input devices.

The following sections describe the operator input devices that were identified as being common to the systems under investigation. The possible hardware configurations of the devices are discussed, and their required functions are listed in pseudo-code. Sample code listings of some of the header files for modules used in actual systems are given in Appendix C and Appendix E.

5.2. Keyboard

An alpha-numeric keyboard is used primarily for the selection of MMI elements by the input of their identification tags. The spacebar is used in conjunction with the joystick for PPI element selection.

5.2.1. Keyboard Configurations

The types of keyboards used can vary between the systems, but part of the standardisation process on the hardware side includes standardising the system keyboard. The standard keyboard used to date is the STORM keyboard that has a SKINT decoder PCB to convert its output to RS232 to interface to the system. The RS232 keyboard codes received by the system are converted into MBII messages which are received by the software keyboard driver, and have to be decoded into normal IBM keyboard codes for use by the rest of the program. In the early prototype systems, and for debugging, a normal IBM keyboard is used, and is plugged into the keyboard input socket of the PC. The normal PC keyboard interrupt is then used to read the keypresses in IBM ASCII and code values. In later PC prototype systems, it is sometimes necessary to plug the STORM keyboard into a PC serial port. In this case, the serial port's interrupt is used to receive the keyboard codes from the keyboard's SKINT PCB, which must again be decoded into IBM keyboard codes.

5.2.2. Keyboard Functions

The following functions are the keyboard device driver functions used to perform the keyboard interface operations.

i) KEYBOARD_Initialise()

This function is required to initialise the keyboard as specified in the input device configuration file.

ii) KEYBOARD_IRQHandler()

This is the interrupt service routine for hardware configurations that use the IBM keyboard input. The routine obtains the ASCII and code values of a keypress, and calls the 'Events' module to store the result in the relevant event queue.

iii) KEYBOARD_MapSkintCode()

This function is required to match a character received from the STORM keyboard's SKINT PCB to its associated IBM keyboard key using a lookup table.

5.3. Joystick

A joystick is used primarily for navigating around the PPI to determine the positions of the system elements, and is also used in conjunction with the keyboard and the softkeys for the selection of MMI elements. In a degraded mode of operation, if the touchpanel and softkeys fail, the joystick is allowed outside the PPI for the selection of soft-buttons and other operations in conjunction with the keyboard spacebar.

5.3.1. Joystick Configurations

The types of joysticks, or positioning devices, used in different systems, and for prototyping and debugging, can vary. In most of the final systems, the

standard hardware velocity joystick is used, which sends messages to the software driver via MBII messages, which must then be decoded into positional information. In the PC prototype systems, and for debugging, a PC mouse is used instead of the system joystick. This mouse is plugged into one of the PC serial ports, and an RS232 interrupt is generated for events such as mouse movement or button pushes. For PC prototypes, the mouse buttons are used instead of the softkeys and touchpanel to generate button and toteline events, by positioning the mouse over the required display object to be activated, and pushing the button. In the actual system, the softkeys and keyboard spacebar are used in conjunction with the joystick position to activate events.

5.3.2. Joystick Functions

The following functions are the joystick device driver functions used to perform the joystick interface operations.

i) JOYSTICK_Initialise()

This function is required to initialise the joystick as specified in the configuration file.

ii) JOYSTICK_IRQHandler()

This is the interrupt service routine for hardware configurations that use the serial port of a PC. The routine obtains the co-ordinates of the joystick's movement, and calls the 'Events' module to store the result in the relevant event queue.

iii) JOYSTICK_GetPosition()

This function is required to get the present pixel position of the joystick. This position is stored in the scenario database in NDCs to be used for event matching to displayed elements such as soft-buttons, and for displaying the current position in the cursor information box.

iv) JOYSTICK_SetPosition()

This function is required to force the joystick to the specified position, for example if joystick movement is restricted to certain areas of the display.

v) JOYSTICK_Show()

This function is required to set a variable allowing the cursor to be displayed each time its position in the scenario database is updated.

vi) JOYSTICK_Hide()

This function is required to set a variable to prevent the cursor being displayed each time its position in the scenario database is updated.

5.4. Softkeys

Softkeys are used for the selection of MMI operations, and are placed at positions around the monitor as required by the particular system. The display positions of the graphical soft-buttons match the positions of the softkeys in order to provide a match for operations.

5.4.1. Softkey Configurations

Softkeys are not usually used in the early PC development and prototype systems, but only in the final operator console. In the PC environment, the IBM keyboard function keys are used to emulate the softkeys, and a match must be done between the function keys and the displayed button positions using a lookup table. In the odd case that the softkeys are used in the PC system, they are plugged into the PC serial port, and the RS232 interrupt is used to receive messages, which must also be decoded.

5.4.2. Softkey Functions

The following functions are the softkey device driver functions used to perform the softkey interface operations.

i) SOFTKEYS_Initialise()

This function is required to initialise the softkeys as specified in the configuration file.

ii) SOFTKEYS_IRQHandler()

This is the interrupt service routine for the hardware configuration that uses the serial port of a PC. The routine obtains the keycode of a keypress, and calls the 'Events' module to store the result in the relevant event queue.

iii) SOFTKEYS_MatchButton()

This function is required to match a softkey code to a displayed button position using lookup tables. A different lookup table is needed for each different hardware configuration (IBM function keys, softkeys plugged into PC serial port), as the codes received may be different. The result of the lookup table match is passed to the 'Events' module to be stored in the relevant event queue.

5.5. TouchPanel

A touchpanel is used as a one-touch positional input device to activate MMI operations, and to select displayed MMI elements.

5.5.1. TouchPanel Configurations

In the PC configuration, the touchpanel is plugged into a PC serial port and messages are received as an RS232 interrupt. In the final systems, the

messages are received on the multibus. In either configuration, the decoding of the message is the same, as the same number of bytes makes up the messages.

5.5.2. TouchPanel Functions

The following functions are the touchpanel device driver functions used to perform the touchpanel interface operations.

i) TOUCHPANEL_Initialise()

This function is required to initialise the touchpanel as specified in the configuration file.

ii) TOUCHPANEL_Calibrate()

This function is required to calibrate the touchpanel so that touchpanel positional events can be decoded into display screen pixel positions. The function is called at initialisation if the configuration file indicates that calibration is required, and during the system's maintenance mode of operation if the operator indicates that recalibration of the touchpanel is required.

iii) TOUCHPANEL_SetMode()

This function is required to change the mode of operation of the touchpanel. The two modes of operation of the touchpanels used are either the single touch mode, where the cursor moves to the position touched and the finger must be raised before touching the screen again for the next event, or the stream mode, where the finger can be dragged across the screen and the cursor will follow.

iv) TOUCHPANEL_IRQHandler()

This is the touch panel interrupt service routine that checks each byte received from the touch panel driver card, checks that the messages are

correctly received, and decodes the message into both a pixel and an NDC position. The result is sent to the Events module to be stored in the relevant event queue.

5.6. Operator Input Device Event Queues

The interface between the operator input devices and the system logic is by means of a series of event queues. Each time an input device event occurs, that event is decoded by an event module or object, and the result is placed in the appropriate event queue. For example, if the position of a push by the operator on the touchpanel matches the position of a particular button, then that button's number would be placed in the button queue.

Thus, the types of common event queues that were identified are:

1) **Soft-button Event Queue**, which holds the number of the soft-button position at which an event has occurred, independent of what operator input device generated the event.
2) **Toteline Event Queue**, which holds the number of the toteline in which an event has occurred, independent of what operator input device generated the event.
3) **PPI Item Event Queue**, which holds the identifier of the selected PPI item, independent of what operator input device was used to select that item.

The 'Events' module or object has the function of decoding and matching operator input device events, obtained by interrupts from the devices, into events that match MMI elements meaningful to the system. Types of matchings required for the various combinations of platforms include:

1) Matching IBM keyboard function key events to soft-button numbers to place in the button queue. This is required in the prototype PC systems where the softkey hardware is not yet available, or in a degraded mode of operation where the softkeys may have failed.
2) Matching a softkey event to a soft-button position to place that button's

number in the button queue.

3) Matching a joystick event position to either:
   a) inside a soft-button to place that button's number in the button queue,
   b) inside a toteline to place that toteline's number in the toteline queue,
   c) the position of an item inside the PPI to place that item selected in the PPI queue.

4) Matching a touchpanel event position to either:
   a) inside a soft-button to place that button's number in the button queue,
   b) inside a toteline to place that toteline's number in the toteline queue,
   c) the position of an item inside the PPI to place that item selected in the PPI queue.

6. SYSTEM CONFIGURATION FILES

System configuration or system setup files are used to enable precompiled software to run on a series of hardware platforms with different system configurations. The configuration files are used to specify the type of graphics display used, and the types of operator input devices used. The configuration files are text files that are read at program startup, to indicate to the program which set of code to run to interface to the hardware.

6.1. Operator Input Device Configuration Files

To speed up the prototyping process, as well as enabling the system software to run on different platforms, the operator input device interface code can be written for different platforms, the software compiled, and only at run-time need the software know what platform is being used as to what set of interface code to run. This is done using configuration files that indicate to the software which platform it is currently running on. These configuration files are read by the software at start-up, and the relevant operator input device interface is used.

The input device configuration files also hold information about the types of input devices to be used, not only their type of interface to the system. For instance, the softkeys may not be present in the prototyping system, so use is made of the function keys on a normal IBM keyboard. Thus, the configuration file is used to indicate to the system that it must interpret function key codes as softkeys, instead of looking for the softkeys themselves.

Figure 16 shows a sample operator input device configuration file, giving the options for the devices to be plugged into COM1 or COM2 of a PC, or to be used in a MBII system, the options for types of softkey and keyboard codes to indicate to the program which lookup tables to use, whether or not touchpanel calibration is required, and whether a normal or extended keyboard is to be used.

```
/*********************************************************/
/*      OPERATOR INPUT DEVICE SYSTEM SETUP FILE          */
/*      -Remove ! from in front of required option       */
/*      -Place ! in front of non-required options        */
/*      -Place a , between options in each list          */
/*      -Place a ~ at end of each options list           */
/*********************************************************/
/* Possible TouchPanel Inputs: Comport 1 OR Comport 2 OR Multibus II */
!TPCOM1,!TPCOM2,TPMBII~
/* Calibrate the TouchPanel */
TPCAL~
/* Possible Softkey Inputs: Comport 1 OR Comport 2 OR Multibus II */
!SKCOM1,!SKCOM2,SKMBII~
/* Possible Softkey Codes: IBM Keyboard OR Skint RS232 OR Skint IBM Codes */
!SKIBMKB,!SKSKINTRS232,SKSKINTIBM~
/* Possible Joystick Inputs: Comport 1 OR Comport 2 OR Multibus II */
!JSKCOM1,!JSKCOM2,JSKMBII~
/* Possible Keyboard Inputs: Comport 1 OR Comport 2 OR Multibus II */
!KBIBM,!KBCOM1,!KBCOM2,KBMBII~
/* Possible Keyboard Type: Present AND Extended OR Not Extended */
KBPRESENT,EXTEDKB~
```

Figure 16: Sample operator input device configuration file.

6.2. Graphics Platform Configuration Files

The graphics configuration files are used slightly differently to the input device configuration files. Unfortunately, the distinction between the different graphical hardware platforms cannot be made only at run-time, as the graphical code needs to include different graphics libraries depending on the graphics platform to be used. This means that the graphical code has to be recompiled and relinked to the rest of the code before it can run on a new platform.

Thus, the graphical configuration file takes the form of information such as the screen size to be used, and other information that the software only needs to know at run-time.

Figure 17 shows a sample graphics configuration file, giving the options for the types of graphics configurations available. These options include whether VGA or RGI graphics is to be used, whether a 14 inch or 19 inch monitor size is to be used, and what the orientation of the display screen is (landscape, portrait left or portrait right).

```
/*********************************************************/
/*             GRAPHICS SYSTEM SETUP FILE                */
/* -Remove ! from in front of required options           */
/* -Place ! in front of non-required options             */
/* -Place a , between options in each list               */
/* -Place a ~ at end of each options list                */
/*********************************************************/
/* Possible Graphics Options: VGA OR RGI */
!VGA,!RGI~
/* Possible Screen Sizes: 14inch OR 19inch */
!14,19~
/* Possible Screen Orientation: Landscape OR Portrait Left OR Portrait Right */
!LAND,!PL,PR~
```

Figure 17: Sample graphics system configuration file.

7. STATE-MACHINE DRIVEN USER-INTERFACES

7.1. Classic State-Machines

A state-machine is an event driven process, where an event causes an action or a series of actions to occur, and the state-machine changes to a new state as a result of the event [19]. Figure 18 shows a classic state-machine process.

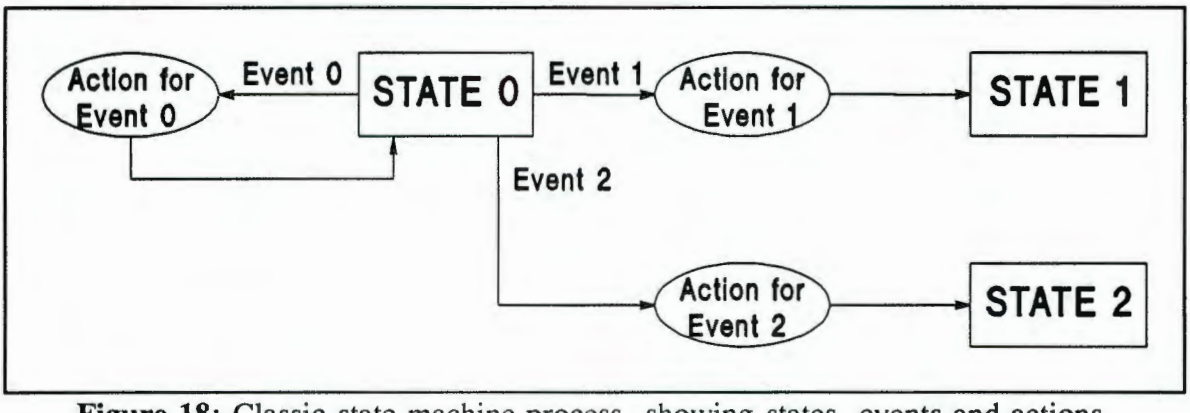

Figure 18: Classic state-machine process, showing states, events and actions.

A real-time environment includes many processes that can be active concurrently [20]. User-system interactions may be viewed as events occurring in this real-time environment [20]. A control process, in the form of a state-machine, is required to co-ordinate these events. The interaction between the program and the user is an event-driven process, in which the events correspond to actions performed by the user via input devices [5]. The use of user-interface state-machines provides for easier and faster iterative development of the interface as requirements change following user comments of prototypes [24].

7.2. System State-Machines

The whole system operation is controlled by a series of state-machines. The two major state-machines are the logic-and-control state-machine, and the user-interface state-machine. The logic-and-control state-machine is used to interact with the system sensors to control their states dependent on operator inputs. The user-interface itself is state-machine driven, the events being operator inputs from the soft-button menus and sub-menus.

7.3. User-Interface State-Machine

The user-interface state-machine code is written in such a way that a state table is created in the form of a 2-dimensional matrix, where the matrix rows represent the current state, and the matrix columns represent the possible events available in the current state. The displayed menus and submenus of soft-buttons represent the current states, and the events are the actual selection of the buttons themselves. The entry in the particular matrix cell that matches the current state and event, is a pointer to the action that must be performed for that particular state and event. Once the action has been performed, the state-machine may change to a new state. The state table is illustrated in Figure 19.

STATE (present menu displayed)	EVENT (button push)	ACTION (function to call)	NEXT STATE (new menu to display)
MAIN MENU	Button 0	Select display item	Main Menu
	Button 1	Draw menu 1	Menu 1
	Button 2	Draw menu 2	Menu 2
	Button 3	Draw menu 3	Menu 3
	Button 4	Draw menu 4	Menu 4
	Button 5	Draw menu 5	Menu 5
	Button 6	Draw menu 6	Menu 6
	Button 7	Draw menu 7	Menu 7
MENU 1	Button 0	Select display item	Menu 1
	Button 1	Action for button 1	Menu 1
	Button 2	Action for button 2	Menu 1
	Button 3	Action for button 3	Menu 1
	Button 4	Action for button 4	Menu 1
	Button 5	Action for button 5	Menu 1
	Button 6	Action for button 6	Menu 1
	Button 7	Draw main menu	Main Menu
MENU 7	Button 0	Select display item	Menu 7
	Button 1	Action for button 1	Menu 7
	Button 2	Action for button 2	Menu 7
	Button 3	Action for button 3	Menu 7
	Button 4	Action for button 4	Menu 7
	Button 5	Action for button 5	Menu 7
	Button 6	Action for button 6	Menu 7
	Button 7	Draw main menu	Main Menu

Figure 19: State table matrix, showing menus, buttons and action functions.

Appendix A gives coded examples of the state table, showing the 2-d array of states (menus) and events (buttons). Each array element is a pointer to a button which must first be initialised and then placed in the array.

8. PRACTICAL IMPLEMENTATION OF STANDARD DESIGN ELEMENTS

Much work in the field of MMI design has been undertaken over the years within UEC Projects. Most of the Command and Control systems produced by the company have as their main component a user-interface that controls the whole system. This interface is the only point of contact between the operator and the system. For ease of operation and training, it was decided that there was a need to standardise certain aspects of the designs so as to prevent as much repetition of coding as possible.

8.1. Design Process

The design process usually entails the following steps:

1) Design and layout of the screen together with the MMI menu and submenu functions available to the operator for controlling the system. This is done using a drawing or draughting package, and forms part of the preliminary design document together with a description of the system operation. This document is discussed with the user, and an initial feedback on the MMI design can be obtained from the user at this early stage before any coding takes place.

2) Initial coding of the MMI design as agreed upon from the preliminary design document is then done. This forms the initial prototype and is usually coded for operation in a normal PC environment using VGA graphics and PC serial ports for operator input devices. User comments on layout and colours can be obtained using this prototype while coding is taking place for the final hardware platform.

3) Coding of the system logic state-machine runs in parallel to the graphics coding, and can be used on the VGA prototype system to gain user feedback on how they feel about the system interaction and response time.

4) The code is transferred to the final hardware platform, having been compiled with the required graphics drivers.

8.2. Projects Undertaken

Projects undertaken to date using the design approaches and standard MMI elements and input devices discussed in this report, include many prototype systems, and two large actual project systems that are currently functional.

In all project designs, an object-oriented design approach is used. For C++ designs, this can be used directly using C++ classes, inheritance, and other aspects of the C++ language that are directly related to object-oriented design. For designs coded in C, object orientation is not built into the language, but can be achieved using structures and modules to represent the classes.

8.2.1. Prototype Systems

Many VGA prototype MMI systems have been designed to obtain user comments before final design, and have also been used as portable demonstration systems to show the company's capabilities. Figure 20 shows a photograph of a typical VGA prototype configuration, with the display layout and colour coding clearly visible.

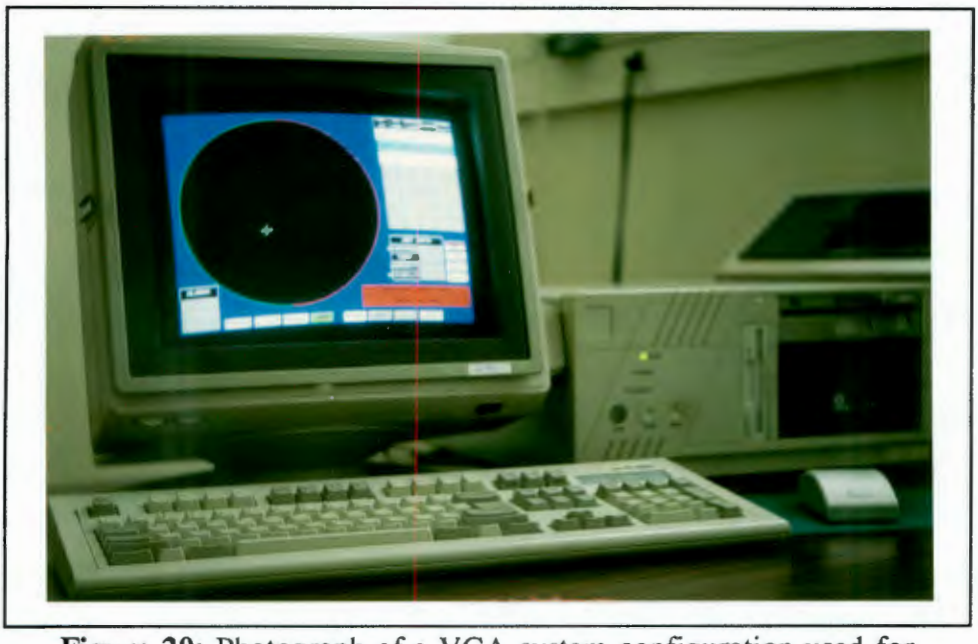

Figure 20: Photograph of a VGA system configuration used for prototyping and development.

8.2.2. Designation Assembly (DA)

This is a system written in Borland C++ that was able to make direct use of the MMI elements designed here by coding them into classes and making use of inheritance to create a truly object-oriented design. This system is currently being installed for operator training courses, having been through the complete design and development stages, including many designer-user demonstrations to obtain user feedback on the screen layout and displayed information. Sample C++ code of some of the standard object header files used in this design is given in Appendix D and Appendix E. Figure 21 shows a photograph of the final DA operator's console, which required a portrait orientation of the display, with the display layout and colour coding clearly visible.

Figure 21: Photograph showing the Designation Assembly operator console.

8.2.3. System Resources Control Unit (SRCU)

This is a system written in C that was able to make use of the MMI elements designed here by coding them into structures and modules rather than C++ classes. The system is written in C as against C++, as at present there is only a C compiler available for the graphics platform used. This system is currently in the concept demonstration phase and has undergone two designer-user demonstration sessions to obtain user feedback on screen layouts and displayed information. Sample C code of some of the standard object header files used in this design is given in Appendix B and Appendix C. Figure 22 shows photographs of the final SRCU operator's console, with the display layout and colour coding clearly visible. The SRCU operator's console makes use of two monitors to convey different sets of information to the operator, the lower screen obviously holding the information most important to the operator.

9. CONCLUSIONS AND RECOMMENDATIONS

This thesis is limited to an overview of the topic of system standardisation. The scope that it covers by no means describes the whole problem and all the solutions. As can be seen from the text, only certain aspects of the problem have been discussed.

The requirements of this investigation were to standardise graphical and operator input device modules, with the aim of being able to transfer the code easily between different hardware platforms. The ability to transfer the software between different platforms has frequently been demonstrated with the MMI work done to date. Examples of this transfer capability were discussed in the body of this report. A particular advantage of the transfer capability is being able to prototype at low cost on PC platforms.

The system design work carried out to date has largely been limited to the use of the languages Pascal, Borland C, and Borland C++. These limitations have been due to the hardware platforms and real-time operating environments used, the design tools available, and to the customer's preferences. These languages may not be the most efficient for rapid, reusable MMI design. It is recommended that further work be done to investigate the possible use of other software packages, such as Visual C++ and Windows. This investigation needs to look not only at the ease of system design, but must also focus on the quality of MMI obtainable in relation to the customer's requirements.

It is also recommended that further work be carried out in the area of observing the operators using the systems in their actual operating environment. This is to gain insight into whether or not the requirements and comments obtained during the prototyping sessions are in actual fact optimal for the operating environment. Since user-interface design is an iterative process, these designs can be constantly modified as requirements change and new systems are developed.

10. REFERENCES

1. DOD-STD-2167A **Defence System Software Development.** 1987.
2. Sommerville, Ian **Software Engineering.** Addison-Wesley, 1989. pp.109-121, pp.257-287, pp.352-359.
3. Myers, Brad A. **User-Interface Tools: Introduction and Survey.** IEEE Software, January 1989. pp.15-23.
4. Tanaka, Toshiaki; Eberts, Ray E.; Salvendy, Gavriel **Consistency of Human-Computer Interface Design: Quantification and Validation.** Human Factors, vol 33, December 1991. pp.653-676.
5. Foley, James D. and Van Dam, Andries **Fundamentals of Interactive Computer Graphics.** Addison-Wesley, 1984.
6. Stankovic, John A. and Ramamritham, Krithi **Hard Real-Time Systems.** IEEE Computer Society, 1988. pp.1-3.
7. Freiden, David R. (Ed) **Principles of Naval Weapon Systems.** Naval Institute Press, 1985. pp.561-583.
8. Huckle, Barbara **The Man-Machine Interface: Guidelines for the design of the End-User/System Conversation.** Savant Institute, 1981.
9. Tracz, Will **RMISE Workshop on Software Reuse Meeting Summary.** In Software Reuse: Emerging Technology, IEEE Computer Society, 1988. pp.42-43.
10. Agresti, William W.; McGarry, Frank E. **The Minnowbrook Workshop On Software Reuse: A Summary Report.** In Software Reuse: Emerging Technology, IEEE Computer Society, 1988. p35.
11. Agresti, William W. **New Paradigms for Software Development.** IEEE Computer Society, 1986. Prototyping articles: p6, p37.
12. Benimoff, Nicholas I. and Whitten, William B. **Human Factors Approaches to Prototyping and Evaluating User Interfaces.** AT&T Technical Journal, vol 68, September/October 1989. pp.44-47.
13. Carey, T.T and Mason, R.E.A **Information System Prototyping: Techniques, Tools, and Methodologies.** In New Paradigms for Software Development, IEEE Computer Society, 1986. pp.48-57.

14. Preece, Jenny and Keller, Laurie **Human Computer Interaction.** Prentice Hall, 1990.

15. Scharer, Laura **The Prototyping Alternative.** In New Paradigms for Software Development, IEEE Computer Society, 1986. pp.59-68.

16. Rice, John F. **Display Color Coding: 10 Rules of Thumb.** IEEE Software, vol 8, January 1991. pp.86-88.

17. Shneiderman, Ben **Designing the User-Interface: Strategies for Effective Human Computer Interaction.** 2nd Ed. Addison-Wesley, 1992.

18. Sutcliffe, Alistair **Human Computer Interface Design.** MacMillan, 1988.

19. Lane, Malcolm G. **Data Communications Software Design.** pp.278-282.

20. Kuo, Feng-Yang and Karimi, Jahangir **User Interface Design from a Real Time Perspective.** Communications of the ACM, vol 31, December 1988. pp.1456-1466.

21. Maddix, Frank **Human-Computer Interaction, Theory and Practice.** Ellis Horwood. pp.250-255.

22. Brown, C. Marlin "Lin" **Human-Computer Interface Design Guidelines.** Ablex Publishing Corporation, 1989.

23. Lange, Beth M. and Moher, Thomas G. **Some Strategies of Reuse in an Object-Oriented Programming Environment.** In "Wings for the Mind" Conference Proceedings, CHI '89, ACM Press. pp.69-73.

24. Wellner, Pierre D. **Statemaster: A UIMS based on Statecharts for Prototyping and Target Implementation.** In "Wings for the Mind" Conference Proceedings, CHI '89, ACM Press. pp.177-182.

25. Mulligan, Robert M.; Alton, Mark W.; Simkin, David K. **User Interface Design in the Trenches: Some Tips on Shooting from the Hip.** In "Reaching Through Technology", Conference Proceedings, CHI '91, ACM Press. pp.232-236.

26. Rumbaugh, James; Blaha, Michael; Premerlani, William; Eddy, Frederick; Lorensen, William **Object-Oriented Modeling and Design.** Prentice Hall, 1991.

27. MIL-STD-1472D **Human Engineering Design Criteria for Military Systems, Equipment and Facilities.** 14 March 1989.

11. BIBLIOGRAPHY

a) Ivanov, D.A., Savel'yev, V.P., Shemansky, P.V. **Fundamentals of Tactical Command and Control (A Soviet View).** Moscow, 1977.

b) Newman, William M. and Sproull, Robert F. **Principles of Interactive Computer Graphics.** 2nd Ed. McGraw-Hill, 1983. pp.443-478.

c) Damodaran, L.; Simpson, A.; Wilson, P. **Designing Systems for People.** National Computing Centre, 1980.

d) Grandjean, E. **Fitting the Task to the Man: An ergonomic approach.** Taylor and Francis, 1987.

e) Tracz, Will **Software Reuse: Emerging Technology.** IEEE Computer Society, 1988.

f) Meyer, Bertrand **Reusability: The Case for Object-Oriented Design.** In Object-Oriented Computing, Volume 2: Implementations. IEEE Computer Society. pp.38-40.

g) Gomaa, Hassan and Scott, Douglas B.H. **Prototyping as a Tool in the Specification of User Requirements.** In New Paradigms for Software Development, IEEE Computer Society, 1986. pp.69-77.

h) Ashby, Philip **Ergonomics Handbook 2: Human Factors Design Data, Displays and Controls.** Design Institute, 1978.

i) Dinan, John A. **Human-interface rules reduce test-program operator errors.** Electronic Design News (EDN), vol 37, February 3, 1992. pp.95-98.

j) Edge, Raimund K. and Stary, Christian **Designing Maintainable, Reusable Interfaces.** IEEE Software, vol 9, November 1992. pp.24-32.

k) Hoadley, Ellen D. **Investigating the Effects of Colour.** Communications of the ACM, vol 33, February 1990. pp.120-125.

l) Jacobson, Bob **The Ultimate User Interface.** Byte, vol 17, April 1992. pp.175-182.

m) Farber, James M. **The AT&T User-Interface Architecture.** AT&T Technical Journal, vol 68, September/October 1989. pp.9-11.

n) Hartson, H. Rex; Hix, Deborah; Kraly, Thomas M. **Developing Human-Computer Interface Models and Representation Techniques.** Software:

Practice and Experience, vol 20, May 1990. pp.425-457.

o) Kantorowitz, Eliezer and Sudarsky, Oded **The Adaptable User Interface.** Communications of the ACM, vol 32, November 1989. pp.1352-1358.

p) Good, Michael D.; Whiteside, John A.; Wixon, Dennis R. **Building a User-Derived Interface.** Communications of the ACM, vol 27, October 1984. pp.1032-1043.

q) Dillard, Robin A. **Using Data Quality Measures in Decision-Making Algorithms.** IEEE Expert, 1992. p63.

r) Fischer, Gerhard **Human-Computer Interaction Software: Lessons Learned, Challenges Ahead.** IEEE Software, January 1989. pp.44-52.

s) Day, Mary Carol **Designing the Human Interface: an Overview.** AT&T Technical Journal, vol 68, September/October 1989. pp.2-8.

t) Neal, Lisa Rubin **A System for Example-Based Programming.** In "Wings for the Mind" Conference Proceedings, CHI '89, ACM Press. pp.63-68.

u) Gutierrez, Oscar **Prototyping Techniques for Different Problem Contexts.** In "Wings for the Mind" Conference Proceedings, CHI '89, ACM Press. pp.259-264.

v) Tetzlaff, Linda and Schwartz, David R. **The Use of Guidelines in Interface Design.** In "Reaching Through Technology" Conference Proceedings, CHI '91, ACM Press. pp.329-333.

w) Chien, Jen-Hsien; Fu, Sheng-Tsai; Horowitz, Ellis **RPP: A System for Prototyping User Interfaces.** In "Reaching Through Technology" Conference Proceedings, CHI '91, ACM Press. pp.419-420.

x) Bier. Eric A. and Pier, Ken **Documents as User Interfaces.** In "Reaching Through Technology" Conference Proceedings, CHI '91. ACM Press. pp.443-444.

APPENDIX A

Sample C and C++ user-interface driver state machine code.

ALCSTMCH.C34

BUTSTMCH.CPP

```
/*****************************************************************************/
/* PROGRAM NAME : ALCSTMCH.C34                                               */
/* DESCRIPTION  : Main state machine driver module.                          */
/* AUTHOR       : C.E. Alston                                                */
/* DATE         : 1994                                                       */
/*****************************************************************************/
#include "alcstmch.hpp"

/*===========================================================================*/
/*===                  DEFINE MAXIMUM MENUS AND BUTTONS                   ===*/
/*===========================================================================*/
#define MAX_NO_MENUS       5
#define NO_OF_MENU_BUTTONS 4

/*===========================================================================*/
/*===                       DEFINE ALL MENUS NEEDED                       ===*/
/*===========================================================================*/
typedef enum
{
 MAIN_MENU = 0,
 SUBMENU1  = 1,
 SUBMENU2  = 2,
 SUBMENU3  = 3,
 SUBMENU4  = 4
} tMenus;

/*===========================================================================*/
/*===             DEFINE ALL BUTTONS NEEDED AND THEIR POSITIONS           ===*/
/*===========================================================================*/
typedef enum
{
 /* Main Menu Buttons */
 B_SUBMENU1 = 0,
 B_SUBMENU2 = 1,
 B_SUBMENU3 = 2,
 B_SUBMENU4 = 3,

 /* Return to Main Menu Button */
 B_MAINMENU = 3,

 /* Spare Buttons */
 B_SPARE0 = 0,
 B_SPARE1 = 1,
 B_SPARE2 = 2,
 B_SPARE3 = 3,

 /* Sub-Menu 1 Buttons */
 B_SM1_BUTTON0 = 0,
 B_SM1_BUTTON1 = 1,
 B_SM1_BUTTON2 = 2,

 /* Sub-Menu 2 Buttons */
 B_SM2_BUTTON0 = 0,
 B_SM2_BUTTON1 = 1,
 B_SM2_BUTTON2 = 2,

 /* Sub-Menu 3 Buttons */
 B_SM3_BUTTON0 = 0,
 B_SM3_BUTTON1 = 1,
 B_SM3_BUTTON2 = 2,

 /* Sub-Menu 4 Buttons */
 B_SM4_BUTTON0 = 0,
 B_SM4_BUTTON1 = 1,
 B_SM4_BUTTON2 = 2
} tButtons;

/*==============================*/
/* Define 2-D state table array */
/*==============================*/
static button_type* _Button_array[MAX_NO_ALLOC_MENUS][MAX_NO_ALLOC_MENU_BUTTONS];
/* Store the menu currently displayed */
static tMenus _Present_menu;
/* Array of all possible button positions in ndc's ie: % of screen size */
extern ndc_rectangle button_positions[MAX_NO_BUTTON_POSITIONS];

/*===========================================================================*/
/*===            DECLARE INSTANCES OF ALL BUTTONS ON THE MMI             ===*/
/*===========================================================================*/
/* MAIN MENU BUTTON -- PRESENT IN ALL SUB-MENUS*/
static button_type MainMenu;

/* SPARE BUTTONS */
static button_type Spare0;
static button_type Spare1;
static button_type Spare2;
static button_type Spare3;
```

```
/* MAIN MENU BUTTONS */
static button_type SubMenu1;
static button_type SubMenu2;
static button_type SubMenu3;
static button_type SubMenu4;

/* SUB_MENU 1 BUTTONS */
static button_type SM1Button0;
static button_type SM1Button1;
static button_type SM1Button2;

/* SUB_MENU 2 BUTTONS */
static button_type SM2Button0;
static button_type SM2Button1;
static button_type SM2Button2;

/* SUB_MENU 3 BUTTONS */
static button_type SM3Button0;
static button_type SM3Button1;
static button_type SM3Button2;

/* SUB_MENU 4 BUTTONS */
static button_type SM4Button0;
static button_type SM4Button1;
static button_type SM4Button2;


/* Prototypes of all action functions called by the state machine */
static void MainMenuSelection(void);

/*=============================================================================*/
/*===              DEFINE ALL THE BUTTON ACTION FUNCTIONS                   ===*/
/*=============================================================================*/
/*-------------------------------------------------------*/
/* FUNCTION    :                                         */
/* DESCRIPTION :                                         */
/* INPUTS      :                                         */
/* OUTPUTS     :                                         */
/*-------------------------------------------------------*/
static void MainMenuSelection(void)
{
 ALLOCSTMCH_DrawMenu(MAIN_MENU);
} /* end of function MainMenuSelection */

/*-------------------------------------------------------*/
/* FUNCTION    :                                         */
/* DESCRIPTION :                                         */
/* INPUTS      :                                         */
/* OUTPUTS     :                                         */
/*-------------------------------------------------------*/
static void SpareSelection(void)
{
} /* end of function SpareSelection */


/*=============================================================================*/
/*===      DEFINITIONS OF ALL BUTTON STATE MACHINE CLASS FUNCTIONS         ===*/
/*=============================================================================*/
/*-------------------------------------------------------*/
/* FUNCTION    :                                         */
/* DESCRIPTION : Initialise the button state machine.    */
/* INPUTS      :                                         */
/* OUTPUTS     :                                         */
/*-------------------------------------------------------*/
void ALLOCSTMCH_Initialise(void)
{
 int menu_no = 0;
 int button_no = 0;

 _Present_menu = M_MAIN_MENU;
 for (menu_no = 0; menu_no < MAX_NO_ALLOC_MENUS; menu_no++)
 {
  for (button_no = 0; button_no < MAX_NO_ALLOC_MENU_BUTTONS; button_no++)
  {
   button_pressed[menu_no][button_no] = FALSE;
   button_pressed_count[menu_no][button_no] = 0;
  }
 }
/* MAIN MENU BUTTON -- PRESENT IN ALL SUB-MENUS*/
 BUTTON_Initialise1(&MainMenu, button_positions[B_MAINMENU], B_ACTIVE, MainMenuSelection, "MAIN-MENU",
B_AVAILABLE);

/* SPARE BUTTONS */
 BUTTON_Initialise1(&Spare0, button_positions[B_SPARE0], B_INACTIVE, SpareSelection, "", B_UNAVAILABLE);
 BUTTON_Initialise1(&Spare1, button_positions[B_SPARE1], B_INACTIVE, SpareSelection, "", B_UNAVAILABLE);
 BUTTON_Initialise1(&Spare2, button_positions[B_SPARE2], B_INACTIVE, SpareSelection, "", B_UNAVAILABLE);
 BUTTON_Initialise1(&Spare3, button_positions[B_SPARE3], B_INACTIVE, SpareSelection, "", B_UNAVAILABLE);
```

```
/* MAIN MENU BUTTONS */
 BUTTON_Initialise1(&SubMenu1,  button_positions[B_SUBMENU1],  B_ACTIVE,  SubMenu1Selection,  "SUBMENU1",
B_AVAILABLE);
 BUTTON_Initialise1(&SubMenu2,  button_positions[B_SUBMENU2],  B_ACTIVE,  SubMenu2Selection,  "SUBMENU2",
B_AVAILABLE);
 BUTTON_Initialise1(&SubMenu3,  button_positions[B_SUBMENU3],  B_ACTIVE,  SubMenu3Selection,  "SUBMENU3",
B_AVAILABLE);
 BUTTON_Initialise1(&SubMenu4,  button_positions[B_SUBMENU4],  B_ACTIVE,  SubMenu4Selection,  "SUBMENU4",
B_AVAILABLE);

/* SUB_MENU 1 BUTTONS */
 BUTTON_Initialise1(&SM1Button0,button_positions[B_SM1_BUTTON0],B_ACTIVE,SM1Button0Selection,"BUTTON1",
B_AVAILABLE);
 BUTTON_Initialise1(&SM1Button1,button_positions[B_SM1_BUTTON1],B_ACTIVE,SM1Button1Selection,"BUTTON2",
B_AVAILABLE);
 BUTTON_Initialise1(&SM1Button2,button_positions[B_SM1_BUTTON2],B_ACTIVE,SM1Button2Selection,"BUTTON3",
B_AVAILABLE);

/* SUB_MENU 2 BUTTONS */
 BUTTON_Initialise1(&SM2Button0,button_positions[B_SM2_BUTTON0],B_ACTIVE,SM2Button0Selection,"BUTTON1",
B_AVAILABLE);
 BUTTON_Initialise1(&SM2Button1,button_positions[B_SM2_BUTTON1],B_ACTIVE,SM2Button1Selection,"BUTTON2",
B_AVAILABLE);
 BUTTON_Initialise1(&SM2Button2,button_positions[B_SM2_BUTTON2],B_ACTIVE,SM2Button2Selection,"BUTTON3",
B_AVAILABLE);

/* SUB_MENU 3 BUTTONS */
 BUTTON_Initialise1(&SM3Button0,button_positions[B_SM3_BUTTON0],B_ACTIVE,SM3Button0Selection,"BUTTON1",
B_AVAILABLE);
 BUTTON_Initialise1(&SM3Button1,button_positions[B_SM3_BUTTON1],B_ACTIVE,SM3Button1Selection,"BUTTON2",
B_AVAILABLE);
 BUTTON_Initialise1(&SM3Button2,button_positions[B_SM3_BUTTON2],B_ACTIVE,SM3Button2Selection,"BUTTON3",
B_AVAILABLE);

/* SUB_MENU 4 BUTTONS */
 BUTTON_Initialise1(&SM4Button0,button_positions[B_SM4_BUTTON0],B_ACTIVE,SM4Button0Selection,"BUTTON1",
B_AVAILABLE);
 BUTTON_Initialise1(&SM4Button1,button_positions[B_SM4_BUTTON1],B_ACTIVE,SM4Button1Selection,"BUTTON2",
B_AVAILABLE);
 BUTTON_Initialise1(&SM4Button2,button_positions[B_SM4_BUTTON2],B_ACTIVE,SM4Button2Selection,"BUTTON3",
B_AVAILABLE);

/*============================================================================*/
/*===   INITIALISE THE BUTTON STATE MACHINE ARRAY WITH ALL THE BUTTONS    ===*/
/*============================================================================*/
 /* Main Menu Buttons */
 _Button_array[MAIN_MENU][B_SUBMENU1] = &SubMenu1;
 _Button_array[MAIN_MENU][B_SUBMENU2] = &SubMenu2;
 _Button_array[MAIN_MENU][B_SUBMENU3] = &SubMenu3;
 _Button_array[MAIN_MENU][B_SUBMENU4] = &SubMenu4;

 /* Sub-Menu 1 Buttons */
 _Button_array[SUBMENU1][B_SM1_BUTTON0] = &SM1Button0;
 _Button_array[SUBMENU1][B_SM1_BUTTON1] = &SM1Button1;
 _Button_array[SUBMENU1][B_SM1_BUTTON2] = &SM1Button2;
 _Button_array[SUBMENU1][B_MAINMENU] = &MainMenu;

 /* Sub-Menu 2 Buttons */
 _Button_array[SUBMENU2][B_SM2_BUTTON0] = &SM2Button0;
 _Button_array[SUBMENU2][B_SM2_BUTTON1] = &SM2Button1;
 _Button_array[SUBMENU2][B_SM2_BUTTON2] = &SM2Button2;
 _Button_array[SUBMENU2][B_MAINMENU] = &MainMenu;

 /* Sub-Menu 3 Buttons */
 _Button_array[SUBMENU3][B_SM3_BUTTON0] = &SM3Button0;
 _Button_array[SUBMENU3][B_SM3_BUTTON1] = &SM3Button1;
 _Button_array[SUBMENU3][B_SM3_BUTTON2] = &SM3Button2;
 _Button_array[SUBMENU3][B_MAINMENU] = &MainMenu;

 /* Sub-Menu 4 Buttons */
 _Button_array[SUBMENU4][B_SM4_BUTTON0] = &SM4Button0;
 _Button_array[SUBMENU4][B_SM4_BUTTON1] = &SM4Button1;
 _Button_array[SUBMENU4][B_SM4_BUTTON2] = &SM4Button2;
 _Button_array[SUBMENU4][B_MAINMENU] = &MainMenu;

 ALLOCSTMCH_DrawMenu(MAIN_MENU);

} /* end of function Initialise */

/*============================================================================*/
/*===              FUNCTIONS TO DRAW BUTTONS AND MENUS                     ===*/
/*============================================================================*/
/*----------------------------------------------------------------------------*/
/* FUNCTION    :                                                              */
/* DESCRIPTION : Function to draw a button only if its state has changed.     */
/* INPUTS      :                                                              */
/* OUTPUTS     :                                                              */
```

```
/*-------------------------------------------------------------------------*/
static void ALLOCSTMCH_DrawButton(tMenus menu, tButtons button)
{

 DISPLAYBUTTONS_Update(button, _Button_array[menu][button]);

} /* end of function DrawButton */

/*-------------------------------------------------*/
/* FUNCTION    :                                   */
/* DESCRIPTION : Function to draw a menu of buttons. */
/* INPUTS      :                                   */
/* OUTPUTS     :                                   */
/*-------------------------------------------------*/
static void ALLOCSTMCH_DrawMenu(tMenus menu)
{
 int button = 0;

 _Present_menu = menu;
 for (button = 0; button < MAX_NO_ALLOC_MENU_BUTTONS; button++)
      DISPLAYBUTTONS_SetNewButton(button, _Button_array[menu][button]);

} /* end of function DrawMenu */

/*=========================================================================*/
/*=== MAIN EXECUTION FUNCTION CALLED WHEN OPERATOR GETS A CHANCE TO RUN ===*/
/*=========================================================================*/
/*-------------------------------------------------*/
/* FUNCTION    :                                   */
/* DESCRIPTION : Button state machine driver function. */
/* INPUTS      :                                   */
/* OUTPUTS     :                                   */
/*-------------------------------------------------*/
static void ALLOCSTMCH_ButtonStateMachine(int button_number)
{

 /* MAIN MENUS STATE MACHINE */
 if ((button_number >= 0) && (button_number < MAX_NO_ALLOC_MENU_BUTTONS))
 {
  if (BUTTON_GetActive(_Button_array[_Present_menu][button_number]) == TRUE)
  {
   BUTTON_GetAction(_Button_array[_Present_menu][button_number])();
   DISPLAYBUTTONS_Pushed(button_number);
  }
 }

} /* end of function ButtonStateMachine */
```

```
/*****************************************************************************/
/* PROGRAM NAME : BUTSTMCH.CPP                                               */
/* DESCRIPTION  : Main state machine driver module.                          */
/* AUTHOR       : C.E. Alston                                                */
/* DATE         : 1993                                                       */
/*****************************************************************************/
#ifndef _BUTSTMCH_HPP_
 #include "butstmch.hpp"
#endif

/*===========================================================================*/
/*===                  DEFINE MAXIMUM MENUS AND BUTTONS                   ===*/
/*===========================================================================*/
#define MAX_NO_MENUS       5
#define NO_OF_MENU_BUTTONS 4

/*===========================================================================*/
/*===                      DEFINE ALL MENUS NEEDED                        ===*/
/*===========================================================================*/
typedef enum
{
 MAIN_MENU = 0,
 SUBMENU1  = 1,
 SUBMENU2  = 2,
 SUBMENU3  = 3,
 SUBMENU4  = 4
} tMenus;

/*===========================================================================*/
/*===            DEFINE ALL BUTTONS NEEDED AND THEIR POSITIONS          ===*/
/*===========================================================================*/
typedef enum
{
 /* Main Menu Buttons */
 B_SUBMENU1 = 0,
 B_SUBMENU2 = 1,
 B_SUBMENU3 = 2,
 B_SUBMENU4 = 3,

 /* Return to Main Menu Button */
 B_MAINMENU = 3,

 /* Spare Buttons */
 B_SPARE0 = 0,
 B_SPARE1 = 1,
 B_SPARE2 = 2,
 B_SPARE3 = 3,

 /* Sub-Menu 1 Buttons */
 B_SM1_BUTTON0 = 0,
 B_SM1_BUTTON1 = 1,
 B_SM1_BUTTON2 = 2,

 /* Sub-Menu 2 Buttons */
 B_SM2_BUTTON0 = 0,
 B_SM2_BUTTON1 = 1,
 B_SM2_BUTTON2 = 2,

 /* Sub-Menu 3 Buttons */
 B_SM3_BUTTON0 = 0,
 B_SM3_BUTTON1 = 1,
 B_SM3_BUTTON2 = 2,

 /* Sub-Menu 4 Buttons */
 B_SM4_BUTTON0 = 0,
 B_SM4_BUTTON1 = 1,
 B_SM4_BUTTON2 = 2

} tButtons;

// Array of all possible button positions in ndc's ie: % of screen size
extern ndc_rectangle button_positions[MAX_NO_BUTTON_POSITIONS];

// Store the menu currently displayed
static tMenus _Present_menu = MAIN_MENU;
// Prototypes of all action functions called by the state machine
static void MainMenuSelection(void);

/*===========================================================================*/
/*===              DEFINE ALL THE BUTTON ACTION FUNCTIONS                 ===*/
/*===========================================================================*/
/*---------------------------------------------------*/
/* FUNCTION    :                                     */
/* DESCRIPTION : Action function for main menu button. */
/* INPUTS      :                                     */
/* OUTPUTS     :                                     */
/*---------------------------------------------------*/
```

```
static void MainMenuSelection(void)
{
 ButtonStateMachine->DrawMenu(MAIN_MENU);
} // end of function MainMenuSelection

/*-------------------------------------------------------*/
/* FUNCTION    :                                         */
/* DESCRIPTION :                                         */
/* INPUTS      :                                         */
/* OUTPUTS     :                                         */
/*-------------------------------------------------------*/
static void SpareSelection(void)
{
} // end of function SpareSelection

/*=======================================================================*/
/*===            DECLARE INSTANCES OF ALL BUTTONS ON THE MMI           ===*/
/*=======================================================================*/
/* MAIN MENU BUTTON -- PRESENT IN ALL SUB-MENUS*/
static logical_button MainMenu(button_positions[B_MAINMENU], B_ACTIVE, MainMenuSelection, "MAIN~MENU",
B_AVAILABLE);

/* SPARE BUTTONS */
static logical_button Spare0(button_positions[B_SPARE0], B_INACTIVE, SpareSelection, "", B_UNAVAILABLE);
static logical_button Spare1(button_positions[B_SPARE1], B_INACTIVE, SpareSelection, "", B_UNAVAILABLE);
static logical_button Spare2(button_positions[B_SPARE2], B_INACTIVE, SpareSelection, "", B_UNAVAILABLE);
static logical_button Spare3(button_positions[B_SPARE3], B_INACTIVE, SpareSelection, "", B_UNAVAILABLE);

/* MAIN MENU BUTTONS */
static logical_button SubMenu1(button_positions[B_SUBMENU1], B_ACTIVE, SubMenu1Selection, "SUBMENU1",
B_AVAILABLE);
static logical_button SubMenu2(button_positions[B_SUBMENU2], B_ACTIVE, SubMenu2Selection, "SUBMENU2",
B_AVAILABLE);
static logical_button SubMenu3(button_positions[B_SUBMENU3], B_ACTIVE, SubMenu3Selection, "SUBMENU3",
B_AVAILABLE);
static logical_button SubMenu4(button_positions[B_SUBMENU4], B_ACTIVE, SubMenu4Selection, "SUBMENU4",
B_AVAILABLE);

/* SUB_MENU 1 BUTTONS */
static logical_button SM1Button0(button_positions[B_SM1_BUTTON0], B_ACTIVE, SM1Button0Selection, "BUTTON1",
B_AVAILABLE);
static logical_button SM1Button1(button_positions[B_SM1_BUTTON1], B_ACTIVE, SM1Button1Selection, "BUTTON2",
B_AVAILABLE);
static logical_button SM1Button2(button_positions[B_SM1_BUTTON2], B_ACTIVE, SM1Button2Selection, "BUTTON3",
B_AVAILABLE);

/* SUB_MENU 2 BUTTONS */
static logical_button SM2Button0(button_positions[B_SM2_BUTTON0], B_ACTIVE, SM2Button0Selection, "BUTTON1",
B_AVAILABLE);
static logical_button SM2Button1(button_positions[B_SM2_BUTTON1], B_ACTIVE, SM2Button1Selection, "BUTTON2",
B_AVAILABLE);
static logical_button SM2Button2(button_positions[B_SM2_BUTTON2], B_ACTIVE, SM2Button2Selection, "BUTTON3",
B_AVAILABLE);

/* SUB_MENU 3 BUTTONS */
static logical_button SM3Button0(button_positions[B_SM3_BUTTON0], B_ACTIVE, SM3Button0Selection, "BUTTON1",
B_AVAILABLE);
static logical_button SM3Button1(button_positions[B_SM3_BUTTON1], B_ACTIVE, SM3Button1Selection, "BUTTON2",
B_AVAILABLE);
static logical_button SM3Button2(button_positions[B_SM3_BUTTON2], B_ACTIVE, SM3Button2Selection, "BUTTON3",
B_AVAILABLE);

/* SUB_MENU 4 BUTTONS */
static logical_button SM4Button0(button_positions[B_SM4_BUTTON0], B_ACTIVE, SM4Button0Selection, "BUTTON1",
B_AVAILABLE);
static logical_button SM4Button1(button_positions[B_SM4_BUTTON1], B_ACTIVE, SM4Button1Selection, "BUTTON2",
B_AVAILABLE);
static logical_button SM4Button2(button_positions[B_SM4_BUTTON2], B_ACTIVE, SM4Button2Selection, "BUTTON3",
B_AVAILABLE);


/*=======================================================================*/
/*===   INITIALISE THE BUTTON STATE MACHINE ARRAY WITH ALL THE BUTTONS    ===*/
/*=======================================================================*/
logical_button* button_state_machine::_Button_array[MAX_NO_MENUS][NO_OF_MENU_BUTTONS] =
{
 /* Main Menu */
 {&SubMenu1, &SubMenu2, &SubMenu3, &SubMenu4},
 /* Sub-Menu 1 */
 {&SM1Button0, &SM1Button1, &SM1Button2, &MainMenu},
 /* Sub-Menu 2 */
 {&SM2Button0, &SM2Button1, &SM2Button2, &MainMenu},
 /* Sub-Menu 3 */
 {&SM3Button0, &SM3Button1, &SM3Button2, &MainMenu},
 /* Sub-Menu 4 */
 {&SM4Button0, &SM4Button1, &SM4Button2, &MainMenu},
};
```

```
/*============================================================================*/
/*===              MAIN BUTTON STATE MACHINE DRIVER FUNCTION                 ===*/
/*============================================================================*/
/*----------------------------------------------------*/
/* FUNCTION    :                                      */
/* DESCRIPTION : Button state machine driver function. */
/* INPUTS      :                                      */
/* OUTPUTS     :                                      */
/*----------------------------------------------------*/
void button_state_machine::ButtonStateMachine(int button_number)
{

 /* Check that the button number received is valid */
 if ((button_number >= 0) && (button_number < NO_OF_MENU_BUTTONS))
 {
  /* Only call the action function if the button is available for selection */
  if (_Button_array[_Present_menu][button_number]->Active())
  {
   /* Call the button's action function */
   _Button_array[_Present_menu][button_number]->Action()();
   /* Graphically simulate depressing the button */
   G_Buttons_Ptr->Pushed(button_number);
  }
 }

} //end of function ButtonStateMachine
```

APPENDIX B

Sample listings of header files for standard MMI display element modules written in C for the TMS340 chip.

BUTTON.H34

DISPBTNS.H34

MESSBAR.H34

TOTELINE.H34

TOTE.H34

CURSORBX.H34

```
#ifndef _BUTTON_H34_
#define _BUTTON_H34_
/*****************************************************************/
/* PROGRAM NAME : BUTTON.H34                                     */
/* DESCRIPTION  : Header file containing button class declarations. */
/* AUTHOR       : C.E. Alston                                    */
/* DATE         : 1994                                           */
/* $Log $                                                        */
/*****************************************************************/
#ifndef _GLOBAL_H34_
 #include "global.h34"
#endif

/*-------------------------------------------------------------*/
/* Defines to determine if button is available for selection */
/*-------------------------------------------------------------*/
#define B_INACTIVE FALSE
#define B_ACTIVE   TRUE

/*-----------------------------------------------------------------*/
/* VALID BUTTON STATES USED BY THE DISPLAY SERVER TO DISPLAY BUTTON IN
   CORRECT STATE */
/*-----------------------------------------------------------------*/
#define B_INVISIBLE     c_BLACK
#define B_UNAVAILABLE   c_LIGHTGRAY
#define B_AVAILABLE     c_WHITE
#define B_SELECTED      c_GREEN
#define B_TRUE_SELECTED c_RED
#define B_NOT_NEEDED    c_YELLOW

/* Enum for different button parts */
typedef enum
{
 PART1 = 1,
 PART2 = 2,
 PART3 = 3,
 PART4 = 4
} button_parts;

/*-------------------------------------------------------------------------*/
/* Define a pointer to a function that takes no parameters and returns nothing */
/*-------------------------------------------------------------------------*/
typedef void (*function_pointer)(void);

typedef struct
{
 ndc_rectangle position;
 int state;
 char label[50];
} box_type;

typedef struct
{
 /* Graphical attributes */
 int _No_of_parts;
 ndc_rectangle _Active_area;
 int _Changed;
 box_type _Part1;
 box_type _Part2;
 box_type _Part3;
 box_type _Part4;
 /* Logical attributes */
 int _Active;
 function_pointer _Action;
} button_type;

/* Functions to initialise different buttons */
void BUTTON_Initialise1(button_type* Button, ndc_rectangle position, int active, function_pointer action,
                        char* label_part1, int state_part1);
void BUTTON_Initialise2(button_type* Button, ndc_rectangle position, int active, function_pointer action,
                        char* label_part1, int state_part1,
                        char* label_part2, int state_part2);
void BUTTON_Initialise3(button_type* Button, ndc_rectangle position, int active, function_pointer action,
                        char* label_part1, int state_part1,
                        char* label_part2, int state_part2,
                        char* label_part3, int state_part3);
void BUTTON_Initialise4(button_type* Button, ndc_rectangle position, int active, function_pointer action,
                        char* label_part1, int state_part1,
                        char* label_part2, int state_part2,
                        char* label_part3, int state_part3,
                        char* label_part4, int state_part4);

/* Functions to handle logical button attributes */
int BUTTON_GetActive(button_type* Button);
void BUTTON_SetActive(button_type* Button, int active);
function_pointer BUTTON_GetAction(button_type* Button);
```

```
/* Functions to handle graphical button attributes */
void BUTTON_SetChanged(button_type* Button, int changed);
int BUTTON_GetChanged(button_type* Button);
int BUTTON_GetState(button_type* Button, int part_no);
void BUTTON_SetState(button_type* Button, int state, int part_no);
char* BUTTON_GetLabel(button_type* Button, int part_no);
void BUTTON_SetLabel(button_type* Button, char* label, int part_no);
void BUTTON_Hide(button_type* Button);
void BUTTON_Draw(button_type* Button);
int BUTTON_Inside(button_type* Button, ndc point);

#endif /* of _BUTTON_H34_ */
```

```
#ifndef _DISPBTNS_H34_
#define _DISPBTNS_H34_
/*****************************************************************************/
/* PROGRAM NAME : DISPBTNS.H34                                               */
/* DESCRIPTION  : Header file containing displayed buttons declarations.     */
/* AUTHOR       : C.E. Alston                                                */
/* DATE         : 1994                                                       */
/* $Log $                                                                    */
/*****************************************************************************/
#ifndef _GLOBAL_H34_
 #include "global.h34"
#endif

#ifndef _BUTTON_H34_
 #include "button.h34"
#endif

#define MAX_NO_BUTTON_POSITIONS 24

void DISPLAYBUTTONS_Initialise(int display);
void DISPLAYBUTTONS_SetNewButton(int button_no, button_type* new_button);
void DISPLAYBUTTONS_Update(int button_no, button_type* new_button);
void DISPLAYBUTTONS_Pushed(int button_no);
void DISPLAYBUTTONS_ShortPush(int button_no);
void DISPLAYBUTTONS_Draw(void);

#endif /* of _DISPBTNS_H34_ */
```

```
#ifndef _MESSBAR_H34_
#define _MESSBAR_H34_
/*****************************************************************************/
/* PROGRAM NAME : MESSBAR.H34                                                */
/* DESCRIPTION  : Header file containing message bar class declarations.     */
/* AUTHOR       : C.E. Alston                                                */
/* DATE         : 1994                                                       */
/* $Log $                                                                    */
/*****************************************************************************/
#include "global.h34"

typedef struct
{
 ndc_rectangle _Position;
 char _Present_message[100];
 char _Last_message[100];
 int _Text_colour;
 int _State;
 int _Last_state;
 int _Changed;
} message_bar;

void MESSAGEBAR_Initialise(message_bar* Message, float top_left_h, float top_left_v, float bottom_right_h,
float bottom_right_v,
            char* message, int text_colour, int state);
void MESSAGEBAR_Message(message_bar* Message, char* message);
void MESSAGEBAR_Update(message_bar* Message);

#endif /* of _MESSBAR_H34_ */
```

```
#ifndef _TOTELINE_H34_
#define _TOTELINE_H34_
/*****************************************************************************/
/* PROGRAM NAME : TOTELINE.H34                                               */
/* DESCRIPTION  : Header file containing graphical tote line class declarations. */
/* AUTHOR       : C.E. Alston                                                */
/* DATE         : 1994                                                       */
/* $Log $                                                                    */
/*****************************************************************************/
#include "global.h34"

#define MAX_NO_TOTE_COLUMNS 5

/*-------------------------------------------------------------------*/
/* States a tws target and tote line can be in for display purposes */
/*-------------------------------------------------------------------*/
#define TWS_INVISIBLE   c_BLACK
#define TWS_UNAVAILABLE c_LIGHTGRAY
#define TWS_AVAILABLE   c_WHITE
#define TWS_SELECTED    c_LIGHTCYAN
#define TWS_DESIGNATED  c_GREEN
#define TWS_POSITIONED  c_GREEN
#define TWS_NOT_NEEDED  c_BROWN
#define TWS_EDIT        c_RED

#define TARGET_ID_PARAMETER        0
#define AZIMUTH_RELATIVE_PARAMETER 1
#define RANGE_PARAMETER            2
#define COURSE_PARAMETER           3
#define SPEED_PARAMETER            4
#define NO_PARAMETER               5

typedef struct
{
 ndc_rectangle _Position;
 ndc_rectangle _Element[MAX_NO_TOTE_COLUMNS];
 int _No_of_columns;
 int _Highlight_parameter;
 int _TWS_number;
 char target_id_buffer[20];
 char azimuth_relative_buffer[20];
 char range_buffer[20];
 char course_buffer[20];
 char speed_buffer[20];
} toteline_type;

void TOTELINE_Initialise(toteline_type* toteline, ndc_rectangle line_position, int no_of_columns);
void TOTELINE_SetTWSNumber(toteline_type* toteline, int tws_number);
int TOTELINE_GetTWSNumber(toteline_type* toteline);
void TOTELINE_Highlight(toteline_type* toteline, int paramter, int highlight);
void TOTELINE_EditParameter(toteline_type* toteline, int parameter, char* new_label);
void TOTELINE_Update(toteline_type* toteline);
int TOTELINE_Inside(toteline_type* toteline, ndc event_position);

#endif /* of _TOTELINE_H34_ */
```

```
#ifndef _TOTE_H34
#define _TOTE_H34
/*****************************************************************************/
/* PROGRAM NAME : TOTE.H34                                                   */
/* DESCRIPTION  : Header file containing logical tote class declarations.    */
/* AUTHOR       : C.E. Alston                                                */
/* DATE         : 1994                                                       */
/* $Log $                                                                    */
/*****************************************************************************/
#include "global.h34"
#include "toteline.h34"

#define MAX_NO_TOTE_LINES 8

void TOTE_ScrollWholeToteUp(void);
void TOTE_ScrollToteUp(int empty_line);
int TOTE_CanScrollToteUp(void);
void TOTE_ScrollWholeToteDown(void);
void TOTE_ScrollToteDown(int empty_line);
int TOTE_CanScrollToteDown(void);
void TOTE_DropTWS(int tws_no);

int TOTE_TWSSelected(void);
int TOTE_TWSNotDesignated(void);

void TOTE_Execute(void);

void TOTE_Initialise(void);
void TOTE_AddOrUpdateTarget(int tws_no);
void TOTE_AddNewTarget(int tws_no);
void TOTE_Highlight(int paramter, int highlight);
void TOTE_EditParameter(int parameter, char* new_label);
void TOTE_Update(void);
int TOTE_CheckToteLineEvent(ndc event_position);

#endif /* of _TOTE_H34_ */
```

```
#ifndef _CURSORBX_H34_
#define _CURSORBX_H34_
/*****************************************************************************/
/* PROGRAM NAME : CURSORBX.H34                                               */
/* DESCRIPTION  : Header file containing cursor box declarations.            */
/* AUTHOR       : C.E. Alston                                                */
/* DATE         : 1994                                                       */
/* $Log $                                                                    */
/*****************************************************************************/
#include "global.h34"

void CURSORBOX_Initialise(void);
void CURSORBOX_Update(void);

#endif /* of _CURSORBX_H34_ */
```

APPENDIX C

Sample listings of header files for standard operator input device modules written in C for a PC platform, and C for the TMS340 chip.

KEYBOARD.H

MOUSE.H

SOFTKEYS.H34

EVENTS.H34

```
#ifndef _KEYBOARD_H_
#define _KEYBOARD_H_
/*****************************************************************************/
/* PROGRAM NAME : KEYBOARD.H                                                 */
/* DESCRIPTION  : Keyboard interrupt class.                                  */
/* AUTHOR       : C.E. Alston                                                */
/* DATE         : 1993                                                       */
/* $Log $                                                                    */
/*****************************************************************************/
#include "global.h"
#include "keys.h"

void KEYBOARD_Initialise(void);
void KEYBOARD_Execute(void);
key_type KEYBOARD_MapSkintRS232Keys(unsigned char key);
void KEYBOARD_Reset(void);

#endif /*_KEYBOARD_HPP_*/
```

```
#ifndef _MOUSE_H_
#define _MOUSE_H_
/*****************************************************************************/
/* MODULE NAME : MOUSE.H                                                     */
/* DESCRIPTION : Header file for mouse class.                                */
/* AUTHOR      : C.E. Alston                                                 */
/* DATE        : 1993                                                        */
/* $Log $                                                                    */
/*****************************************************************************/
#include "global.h"

void MOUSE_Execute(void);
void MOUSE_Show(void);
void MOUSE_Hide(void);
pixel MOUSE_PresentPosition(void);
void MOUSE_SetPosition(unsigned char x_count, unsigned char y_count);

#endif /*end of _MOUSE_H_*/
```

```
#ifndef _SOFTKEYS_H34_
#define _SOFTKEYS_H34_
/*****************************************************************************/
/* PROGRAM NAME : SOFTKEYS.H34                                               */
/* DESCRIPTION  : Header file containing the softkey class declarations.     */
/* AUTHOR       : C.E. Alston                                                */
/* DATE         : 1994                                                       */
/* $Log $                                                                    */
/*****************************************************************************/
#ifndef _GLOBAL_H34_
 #include "global.h34"
#endif

void SOFTKEYS_Initialise(void);
int SOFTKEYS_Execute(void);

#endif /* of _SOFTKEYS_HPP_ */
```

```
#ifndef _EVENTS_H34_
#define _EVENTS_H34_
/*****************************************************************/
/* FILE NAME   : EVENTS.H34                                      */
/* DESCRIPTION : Header file for the events class.               */
/* AUTHOR      : C.E. Alston                                     */
/* DATE        : 1994                                            */
/* $Log $                                                        */
/*****************************************************************/
#include "global.h34"
#include "queue.h34"
#include "keys.h34"

/* Functions to add events to the event queues */
void EVENTS_Initialise(void);
void EVENTS_SoftkeyEvent(unsigned char keycode);
void EVENTS_KeyboardEvent(key_type keypressed);
void EVENTS_RS232KeyboardEvent(unsigned char key);
void EVENTS_CheckForEvent(void);
void EVENTS_AllocboxLineSpaceBar(void);
void EVENTS_SoftkeyEventCheck(void);
void EVENTS_JSKEventHandler(pixel position);

 /* Functions to decode what events have occurred */
int EVENTS_ButtonSelected(void);
int EVENTS_AllocboxLineSelected(void);
int EVENTS_TWSNumberSelected(void);
void EVENTS_KeyboardEventCheck(void);
void EVENTS_JoystickEventCheck(void);
int EVENTS_Terminate(void);

key_type EVENTS_KeyboardTest(void);
int EVENTS_SoftkeyTest(void);
ndc EVENTS_JoystickTest(void);

#endif /* of _EVENTS_H34_ */
```

APPENDIX D

Sample listings of header files for standard MMI display element modules written in Borland C++.

G_BUTTON.HPP

L_BUTTON.HPP

MESSBAR.HPP

G_TOTELN.HPP

G_TOTE.HPP

L_TOTE.HPP

```
#ifndef _G_BUTTON_HPP_
#define _G_BUTTON_HPP_
/*****************************************************************************/
/* PROGRAM NAME : G_BUTTON.HPP                                               */
/* DESCRIPTION  : Header file containing graphical button class declarations. */
/* AUTHOR       : C.E. Alston + C. de Villiers                               */
/* DATE         : 1993                                                       */
/*****************************************************************************/
#ifndef _GLOBAL_HPP_
 #include "global.hpp"
#endif

#ifndef _COORDS_HPP_
 #include "coords.hpp"
#endif

#ifndef _ATTRIB_HPP_
 #include "attrib.hpp"
#endif

#ifndef _GTEXTBOX_HPP_
 #include "gtextbox.hpp"
#endif

/*-------------------------------------------------------------------------*/
/* VALID BUTTON STATES USED BY THE DISPLAY SERVER TO DISPLAY BUTTON IN
   CORRECT STATE */
/*-------------------------------------------------------------------------*/
typedef enum {B_INVISIBLE           = c_BLACK,
              B_UNAVAILABLE         = c_LIGHT_GREY,    // c_LIGHTGRAY,
              B_AVAILABLE           = c_WHITE,
              B_SELECTED            = c_GREEN,         // c_LIGHTGREEN,
              B_REL_SELECTED        = c_RED,
              B_NOT_NEEDED          = c_YELLOW,
              B_TEST_IN_PROGRESS    = c_YELLOW,
              B_TEST_PASSED         = c_GREEN,
              B_TEST_FAILED         = c_RED,
              B_TEST_UNKNOWN        = c_CYAN,
              B_CARD_NOT_RESPONDING = c_BLUE} button_states;

/*-----------------------*/
/* GRAPHICAL BUTTON CLASS */
/*-----------------------*/
class graphical_button:public gshape
{
private:
 int _No_of_parts;
 wwindow _Active_area;  // covers the entire button
 gtextbox * _Part[4];   // caters for multipart buttons
public:
 graphical_button();
 graphical_button( wwindow position,
                   char* label_part1, button_states state_part1);
 graphical_button( wwindow position,
                   char* label_part1, button_states state_part1,
                   char* label_part2, button_states state_part2);
 graphical_button( wwindow position,
                   char* label_part1, button_states state_part1,
                   char* label_part2, button_states state_part2,
                   char* label_part3, button_states state_part3);
 graphical_button( wwindow position, char* label_part1, button_states state_part1,
                   char* label_part2, button_states state_part2,
                   char* label_part3, button_states state_part3,
                   char* label_part4, button_states state_part4);
   virtual boolean Visible(const boolean IsVisible);
   button_states State(int part_no = 1);
   void State(button_states state, int part_no = 1);
   const char* Label(int part_no);
   void Label(const char* label, int part_no=1);
   boolean Inside(ndc point);
   ~graphical_button();
}; //end of class graphical_button

#endif //_G_BUTTON_HPP_
```

```
#ifndef _L_BUTTON_HPP_
#define _L_BUTTON_HPP_
/*****************************************************************************/
/* PROGRAM NAME : L_BUTTON.HPP                                               */
/* DESCRIPTION  : Header file containing logical button class declarations.  */
/* AUTHOR       : C.E. Alston                                                */
/* DATE         : 1993                                                       */
/* $Log $                                                                    */
/*****************************************************************************/
#ifndef _GLOBAL_HPP_
 #include "global.hpp"
#endif

#ifndef _G_BUTTON_HPP_
 #include "g_button.hpp"
#endif

/* Enums for adressing parts of buttons */
typedef enum
{
 PART1 = 1,
 PART2 = 2,
 PART3 = 3,
 PART4 = 4
} button_parts;

/*-------------------------------------------------------------*/
/* Defines to determine if button is available for selection */
/*-------------------------------------------------------------*/
const boolean B_INACTIVE = FALSE;
const boolean B_ACTIVE = TRUE;

/*-----------------------------------------------------------------------------*/
/* Define a pointer to a function that takes no parameters and returns nothing */
/*-----------------------------------------------------------------------------*/
typedef void (*function_pointer)(void);

/*---------------------*/
/* LOGICAL BUTTON CLASS */
/*---------------------*/
class logical_button: public graphical_button
{
private:
 boolean _Active;
 function_pointer _Action;

public:
 logical_button();
 logical_button(wwindow position, boolean active, function_pointer action,
                char* label_part1, button_states state_part1);
 logical_button(wwindow position, boolean active, function_pointer action,
                char* label_part1, button_states state_part1,
                char* label_part2, button_states state_part2);
 logical_button(wwindow position, boolean active, function_pointer action,
                char* label_part1, button_states state_part1,
                char* label_part2, button_states state_part2,
                char* label_part3, button_states state_part3);
 logical_button(wwindow position, boolean active, function_pointer action,
                char* label_part1, button_states state_part1,
                char* label_part2, button_states state_part2,
                char* label_part3, button_states state_part3,
                char* label_part4, button_states state_part4);
 boolean Active(void);
 void Active(boolean active);
 function_pointer Action(void);
 ~logical_button();
}; //end of class logical_button

#endif //_L_BUTTON_HPP_
```

```
#ifndef _MESSBAR_HPP_
#define _MESSBAR_HPP_
/*****************************************************************************/
/* PROGRAM NAME : MESSBAR.HPP                                                */
/* DESCRIPTION  : Header file containing message bar class declarations.     */
/* AUTHOR       : C.E. Alston                                                */
/* DATE         : 1993                                                       */
/* $Log $                                                                    */
/*****************************************************************************/
#ifndef _GLOBAL_HPP_
 #include "global.hpp"
#endif

#ifndef _GTEXTBOX_HPP_
 #include "gtextbox.hpp"
#endif

const int MAX_NO_MESSAGES = 10;

class message_bar: public gtextbox
{
private:
 struct message_bar_struct
 {
  colour message_colour;
  colour bar_colour;
  char message[80];
 };

 message_bar_struct _Message_array[MAX_NO_MESSAGES];
 int _Message_displayed;

public:
 message_bar();
 message_bar(double top_left_h, double top_left_v,
             double bottom_right_h, double bottom_right_v,
             char* message, colour text_colour, colour state, boolean visible);
 void Message(char* message, colour text_colour = c_DARK_GREY, colour state = c_WHITE);
 void Error(char* message, colour text_colour = c_DARK_GREY, colour state = c_WHITE);
 void Prompt(char* message, colour text_colour = c_DARK_GREY, colour state = c_WHITE);
 void ScrollUp(void);
 void ScrollDown(void);
 ~message_bar();
}; //end of function message_bar

#endif //of _MESSBAR_HPP
```

```
#ifndef _G_TOTELN_HPP_
#define _G_TOTELN_HPP_
/*****************************************************************************/
/* PROGRAM NAME : G_TOTELN.HPP                                               */
/* DESCRIPTION  : Header file containing graphical tote line class declarations. */
/* AUTHOR       : C.E. Alston + C. de Villiers                               */
/* DATE         : 1993                                                       */
/*****************************************************************************/
#ifndef _GLOBAL_HPP_
 #include "global.hpp"
#endif

#ifndef _ATTRIB_HPP_
 #include "attrib.hpp"
#endif

#ifndef _GTEXTBOX_HPP_
 #include "gtextbox.hpp"
#endif

/*-------------------------------------------------------------------*/
/* States a tws target and tote line can be in for display purposes */
/*-------------------------------------------------------------------*/
typedef enum {TWS_INVISIBLE    = c_BLACK,
              TWS_UNAVAILABLE  = c_LIGHT_GREY,
              TWS_AVAILABLE    = c_WHITE,
              TWS_SELECTED     = c_CYAN,
              TWS_DESIGNATED   = c_GREEN,
              TWS_POSITIONED   = c_GREEN,
              TWS_NOT_NEEDED   = c_YELLOW,
              TWS_EDIT         = c_RED} tws_states;

typedef enum {TARGET_ID_PARAMETER,
              AZIMUTH_TRUE_PARAMETER,
              AZIMUTH_RELATIVE_PARAMETER,
              RANGE_PARAMETER,
              COURSE_PARAMETER,
              SPEED_PARAMETER,
              SENSOR_PARAMETER,
              COMMENT_PARAMETER,
              TWS_STATUS_PARAMETER,
              NO_PARAMETER} target_parameter_type;

const int max_no_tote_columns = 9;
const float tote_element_width = 8.0;

class graphical_tote_line
{
private:
 target_parameter_type _Highlight_parameter;
 int _TWS_number;
 wwindow _Position;
 gtextbox _Element[max_no_tote_columns];
public:
 graphical_tote_line();
 void Position(wwindow line_position);
 void Visible(const boolean NewVisible);
 void FillColour(const colour NewColour);
 void BorderColour(const colour NewColour);
 void TextColour(const colour NewColour);
 void TWSNumber(int tws_number);
 int TWSNumber(void);
 void Highlight(target_parameter_type paramter, boolean highlight);
 void EditParameter(target_parameter_type parameter, char* new_label);
 void SetValues(void);
 ~graphical_tote_line();
}; //end of class graphical_tote_line

#endif //of _G_TOTELN_HPP_
```

```
#ifndef _G_TOTE_HPP_
#define _G_TOTE_HPP_
/*****************************************************************************/
/* PROGRAM NAME : G_TOTE.HPP                                                 */
/* DESCRIPTION  : Header file containing graphical tote class declarations. */
/* AUTHOR       : C.E. Alston + C. de Villiers                               */
/* DATE         : 1993                                                       */
/* $Log $                                                                    */
/*****************************************************************************/
#ifndef _GLOBAL_HPP_
 #include "global.hpp"
#endif

#ifndef _G_TOTELN_HPP_
 #include "g_toteln.hpp"
#endif


class tote_heading:public gbox
// Individual tote column headings are added to _ShapeList
{
  public:
  tote_heading();
 ~tote_heading();
}; //end of class tote_heading

class graphical_tote : public gbox
// Contains an array of graphical_tote_line
{
private:
 tote_heading _Tote_heading;

protected:
 graphical_tote_line _Tote[MAX_NO_TOTE_LINES];
 int _Present_tote_line_selected;

public:
 graphical_tote();
 void Highlight(target_parameter_type paramter, boolean highlight);
 void EditParameter(target_parameter_type parameter, char* new_label);
 void SetValues(void);
 boolean Visible(const boolean NewVisible);
 ~graphical_tote();
}; //end of class graphical_tote

#endif //of _G_TOTE_HPP_
```

```
#ifndef _L_TOTE_HPP
#define _L_TOTE_HPP
/*****************************************************************************/
/* PROGRAM NAME : L_TOTE.HPP                                                 */
/* DESCRIPTION  : Header file containing logical tote class declarations.    */
/* AUTHOR       : C.E. Alston                                                */
/* DATE         : 1993                                                       */
/* $Log $                                                                    */
/*****************************************************************************/
#ifndef _GLOBAL_HPP_
 #include "global.hpp"
#endif

#ifndef _G_TOTE_HPP_
 #include "g_tote.hpp"
#endif

class logical_tote : public graphical_tote
{
private:
 void Initialise(void);
 void SelectTWS(int line_no);
 void DeselectTWS(int line_no);
 void Add(int tws_number);
 void DropTsOffTargets(void);
 void AddNewTargetsToTote(void);
 void CheckToteLineSelection(void);
 void CheckPPITargetSelection(void);
 void FillEmptyToteLines(void);
 int TargetAlreadyInTote(int tws_number);

public:
 logical_tote();
 void ScrollToteUp(int empty_line);
 boolean CanScrollToteUp(int number_of_targets_tracked);
 void ScrollToteDown(int empty_line);
 boolean CanScrollToteDown(int number_of_targets_tracked);
 void DropTWS(int line_no = NONE);

 int TWSSelected(void);
 boolean ToteEmpty(void);

 void Execute(void);
 ~logical_tote();
}; //end of class logical_tote

#endif //of _L_TOTE_HPP_
```

APPENDIX E

Sample listings of header files for standard operator input device modules written in Borland C++.

KEYBOARD.HPP

MOUSE.HPP

SOFTKEYS.HPP

TPNL.HPP

G_EVENTS.HPP

L_EVENTS.HPP

```
#ifndef _KEYBOARD_HPP_
#define _KEYBOARD_HPP_
/*****************************************************************************/
/* PROGRAM NAME : KEYBOARD.HPP                                               */
/* DESCRIPTION  : Keyboard interrupt class.                                  */
/* AUTHOR       : C.E. Alston                                                */
/* DATE         : 1993                                                       */
/* $Log $                                                                    */
/*****************************************************************************/
#ifndef _GLOBAL_HPP_
 #include "global.hpp"
#endif

#ifndef _KEYS_HPP_
 #include "keys.hpp"
#endif

const int keyboard_interrupt_level = 0x09;
const int keyboard_irq_number = 0x01;

const class keyboard
{
private:
 intvect oldvect;

 void Initialise(void);
 void Reset(void);

public:
 keyboard();
 void Execute(void);
 key_type MapSkintRS232Keys(unsigned char key);
 ~keyboard();
}; // end of class keyboard


#endif //_KEYBOARD_HPP_
```

```
#ifndef _MOUSE_HPP_
#define _MOUSE_HPP_
/*************************************************************************/
/* MODULE NAME : MOUSE.HPP                                               */
/* DESCRIPTION : Header file for mouse class.                            */
/* AUTHOR      : C.E. Alston                                             */
/* DATE        : 1993                                                    */
/* $Log $                                                                */
/*************************************************************************/
#ifndef _GLOBAL_HPP_
 #include "global.hpp"
#endif

#ifndef _GDRIVER_HPP_
 #include "gdriver.hpp"
#endif

#define MOUSE_INTERRUPT 0x33

/*-----------------------------------------------*/
/* Prototypes of functions used by other modules */
/*-----------------------------------------------*/
class mouse
{
private:
 boolean _Mouse_displayed;
 boolean _Handler_installed;
 boolean  left_pressed;
 boolean  right_pressed;
 boolean  middle_pressed;
 boolean  left_right_pressed;
 boolean  left_middle_pressed;
 boolean  right_middle_pressed;
 boolean  all_three_pressed;
 pixel    position;

 void Initialise(void);
 void Reset(void);
 void SetXRange(int x1, int x2);
 void SetYRange(int y1, int y2);
 void GetStatus(void);
 void RegisterHandler(word interrupt_mask, void (far *interrupt_func)() );
 boolean LeftButtonPressed(void);
 boolean RightButtonPressed(void);
 boolean LeftRightPressed(void);

public:
 mouse();
 void Execute(void);
 void Show(void);
 void Hide(void);
 pixel PresentPosition(void);
 void SetPosition(pixel position);
 void SetPosition(unsigned char x_count, unsigned char y_count);
 ~mouse();
}; //end of class mouse

#endif //end of _MOUSE_HPP_
```

```
#ifndef _SOFTKEYS_HPP_
#define _SOFTKEYS_HPP_
/*****************************************************************************/
/* PROGRAM NAME : SOFTKEYS.HPP                                               */
/* DESCRIPTION  : Header file containing the softkey class declarations.     */
/* AUTHOR       : C.E. Alston                                                */
/* DATE         : 1993                                                       */
/* $Log $                                                                    */
/*****************************************************************************/
#ifndef _GLOBAL_HPP_
 #include "global.hpp"
#endif

class softkeys
{
public:
 softkeys();
 int MatchButton(unsigned char keycode);
 ~softkeys();
}; //end of class softkeys

#endif //end of _SOFTKEYS_HPP_
```

```
#ifndef _TPNL_HPP_
#define _TPNL_HPP_
/*****************************************************************************/
/* MODULE NAME : TPNL.HPP                                                    */
/* DESCRIPTION : Header file for touch panel driver class.                   */
/* AUTHOR      : C.E. Alston                                                 */
/* DATE        : 1993                                                        */
/* $Log $                                                                    */
/*****************************************************************************/
#include <stdio.h>

#ifndef _GLOBAL_HPP_
 #include "global.hpp"
#endif

#ifndef _GDRIVER_HPP_
 #include "gdriver.hpp"
#endif

typedef struct
{
 int x;
 int y;
} int_pixel;

/*-------------------------------------------*/
/* Commands used to initilaise the touch panel */
/*-------------------------------------------*/
typedef enum
{
 NO_COMMAND,
 RESET_CONTROLLER,
 MODE_POINT,
 MODE_STREAM,
 FORMAT_DECIMAL,
 FORMAT_HEXADECIMAL,
 FORMAT_TABLET
} tp_commands;

class touch_panel
{
private:
 struct
 {
  boolean _Calibrated;
  int_pixel _R_low;
  int_pixel _R_high;
  int_pixel _S_low;
  int_pixel _S_high;
  int_pixel _Delta_r;
  int_pixel _Delta_s;
  int_pixel _Xy_point_pressed;
  boolean _Pressed;
 } _Info;
 FILE *_File;
 boolean _Tp_initialised;
 boolean _Response_ok;
 boolean _No_response_from_TP;

 void WaitForResponse(void);
 void WaitForPush(void);
 int SaveParameters(void);
 int LoadParameters(void);
 int_pixel GetCoordinates(int_pixel marker_position);
 void AdjustGraphicsScreen(void);
 void Reset(void);
 boolean Service(int_pixel position_pressed);
 int Calibrate(boolean calibrate);

public:
 touch_panel();
 void Initialise(void);
 void IRQHandler(byte input_byte);
 void ChangeMode(tp_commands mode);
 ~touch_panel();
}; //end of class touch_panel

#endif //end of _TPNL_HPP_
```

```
#ifndef _G_EVENTS_HPP_
#define _G_EVENTS_HPP_
/*****************************************************************/
/* FILE NAME   : G_EVENTS.HPP                                    */
/* DESCRIPTION : Header file for the graphical events class.     */
/* AUTHOR      : C.E. Alston                                     */
/* DATE        : 1993                                            */
/* $Log $                                                        */
/*****************************************************************/
#ifndef _GLOBAL_HPP_
 #include "global.hpp"
#endif

#ifndef _QUEUE_HPP_
 #include "queue.hpp"
#endif

#ifndef _COORDS_HPP_
 #include "coords.hpp"
#endif

typedef enum
{
 NO_EVENT,
 BUTTON_EVENT,
 TOTELINE_EVENT,
 TWS_EVENT
} event_type;

typedef struct
{
 event_type event;
 int button_number;
 int toteline_number;
 int tws_number;
} event_structure;

class graphical_events
{
private:
 queue<event_structure> _Event_queue;
 pixel _Tp_test_point;

 boolean CheckButtonEvent(ndc event_position);
 boolean CheckToteLineEvent(ndc event_position);
 boolean CheckTWSEvent(ndc event_position);

public:
 graphical_events();
 void TPEventHandler(pixel position);
 void JSKEventHandler(pixel position);
 event_structure GetEvent(void);

 /* Functions for device tests in maintenance mode */
 ndc TouchPanelTest(void);
 ndc JoystickTest(void);

 ~graphical_events();
}; //end of class graphical_events

#endif //end of _L_EVENTS_HPP_
```

```
#ifndef _L_EVENTS_HPP_
#define _L_EVENTS_HPP_
/*****************************************************************/
/* FILE NAME   : L_EVENTS.HPP                                    */
/* DESCRIPTION : Header file for the logical events class.       */
/* AUTHOR      : C.E. Alston                                     */
/* DATE        : 1993                                            */
/* $Log $                                                        */
/*****************************************************************/
#ifndef _GLOBAL_HPP_
 #include "global.hpp"
#endif

#ifndef _G_EVENTS_HPP_
 #include "g_events.hpp"
#endif

#ifndef _QUEUE_HPP_
 #include "queue.hpp"
#endif

#ifndef _KEYS_HPP_
 #include "keys.hpp"
#endif

const int max_id_length = 3;

class events : public graphical_events
{
private:
 queue<int> _Button_event_queue;
 queue<int> _Tote_event_queue;
 queue<int> _TWS_event_queue;
 queue<key_type> _Keyboard_event_queue;
 boolean _Maintenance_mode_requested;
 boolean _Operational_mode_requested;
 boolean _Terminate;
 key_type _Testkey;
 int _Test_softkey;

 void CheckTargetIdSelection(key_type keypressed, boolean check_selection);

public:
 events();
 /* Functions to add events to the event queues */
 void MouseEvent(pixel mouse_position);
 void SoftkeyEvent(unsigned char keycode);
 void KeyboardEvent(key_type keypressed);
 void RS232KeyboardEvent(unsigned char key);
 void CheckForEvent(void);
 void MBIITPEventCheck(void);
 void MBIIJSKEventCheck(void);
 void MBIIKBEventCheck(void);
 void MBIISKEventCheck(void);
 void MBIIEventCheck(void);
 void ToteLineSpaceBarSelection(void);

 /* Functions to decode what events have occurred */
 int ButtonSelected(void);
 int ToteLineSelected(void);
 int TWSNumberSelected(void);
 void KeyboardEventCheck(void);
 void MaintenanceModeRequested(boolean requested);
 boolean MaintenanceModeRequested(void);
 void OperationalModeRequested(boolean requested);
 boolean OperationalModeRequested(void);
 boolean Terminate(void);
 key_type KeyboardTest(void);
 int SoftkeyTest(void);
 ~events();
}; //end of class events

#endif //end of _L_EVENTS_HPP_
```