

T. H. E. SOLUTION LLC
Product Development Consulting

C++ Framework and Toolkit for Control Applications
Base Embedded Device Control Applications
Version 1.1

3/27/00

Copyright © 1999 - 2000 by T. H. E. Solution LLC

Table of Contents

1	INTRODUCTION	1
2	DEFINITION	2
3	DESIGN	5
3.1	Architecture	5
3.2	Detail	12
3.2.1	Application	12
3.2.2	Control Panel	15
3.2.3	Operation Base Class	18
3.2.4	Component List Class	21
3.2.5	Operation Block Linked List	25
3.2.6	Automated Operation Manager	26
3.2.7	Mail Box	28
3.2.8	Device Communications	32
3.2.9	File Manager	48
3.2.10	Console	52
4	CONFIGURATION FILE	59
	APPENDIX A, WINDOW'S CE CONSIDERATIONS	61
	APPENDIX B, KNOWN ISSUES	63

1 INTRODUCTION

This document provides the definition and design for T. H. E. Solutions' Windows based Control Application Framework. This framework is designed to facilitate the quick and correct composition of "control applications". These applications are one of multiple components in a system based around an externally configured, programmed or controlled embedded device(s). A typical overall system is diagrammed in Figure 1.

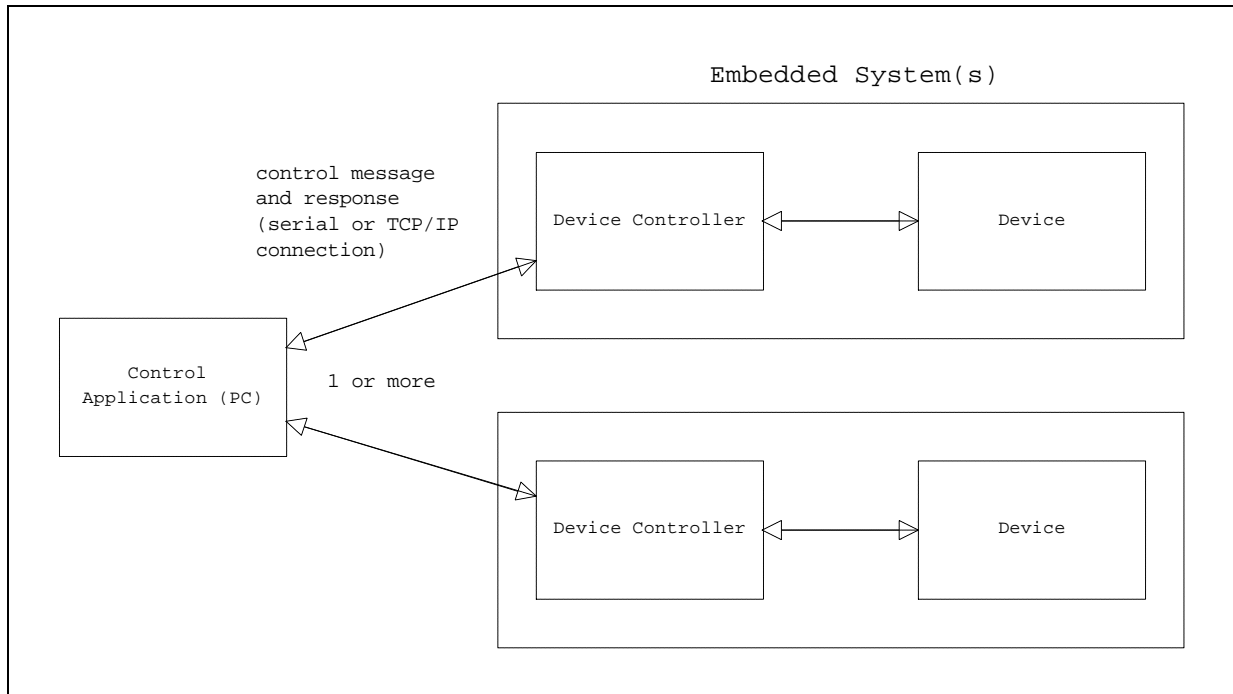


Figure 1, Typical System

The control application provides a user-friendly environment for configuring, programming and/or controlling the embedded system. It is often implemented using some version of Windows on a PC.

2 DEFINITION

The control application framework is designed to facilitate the quick composition of C++ programs that are used to configure and control embedded systems. It is a very specialized framework. This has both advantages and disadvantages. The advantage is that it provides a means to compose C++ control applications with the least amount of overhead and the lowest possible learning curve. The disadvantage is that it does not attempt to provide C++ classes to do everything. It is not a replacement for Borland's OWL or Microsoft's MFC when developing more general-purpose applications.

The framework meets the following general requirements:

1. It supports a "component based model" that facilitates the quick prototyping and development of control applications.
2. This support is tailored for the C++ language, primarily using Microsoft's Visual C++ (5 or 6), in a 32 bit Windows environment. This includes Windows 95/98, Windows NT 4.0, Windows 2000 and Windows CE 2.0/2.01.
3. Where the Windows environment is either difficult or confusing, the framework abstracts away these difficulties.
4. It uses the "mixed" programming model, Windows and traditional multi-tasking events are used together.
5. Where the Windows environment is reasonably straightforward (or well supported by Visual C++), such as dialog boxes, there is no need to build support classes just to "objectify" the Windows API. Instead a generic control application example will provide meaningful examples of the appropriate design and implementation of components to include the use of Window's facilities.
6. Areas that commonly require support by control application developers and/or are substantially different in implementation in the three Win32 environments will be supported with fully implemented classes. The areas supported include:
 - a. Child window console style output
 - b. File input/output
 - c. Communication (currently includes a point to point high reliability serial link, with TAPI based dial-out modem support, and a TCP/IP socket link, with DUN support)
 - d. Application level logging

- e. Application level status reporting (status bar)
 - f. Long term operation status indicator (progress bar)
7. The applications developed have a relatively small "footprint".

We can conceive of a control application as a collection of "components" which interact with each other to provide the desired function set. The idea of a component is similar (and often identical) to that of a classical object. A component is defined as:

1. A collection of data and code that provides a specific function. It provides a block of data and code with high functional cohesion that requires relatively low coupling with other components.
2. It is the base building block for a library of reusable functions.
3. It can be implemented in a module that can be grasped in a reasonable time by a new person that knows the programming language, C++. This in general restricts the source module to ten pages (~ 800 line) including code, comments etc.

In addition a component should be constructed in a manner that eliminates the need to repeat code. The general scheme is to create basic or atomic components, i.e. components that take care of a basic required function, and to use these in more complex components that use several basic operations along with workflow logic etc. to handle a complex operation. Therefore, components in general must be able to invoke and to be invoked by other components. The only limitation is space.

A component should support, but not require, asynchronous operation. This provides the means to take advantage of Win32's multi-threading support when it is advantageous to do so.

Finally, there is a need for a special type of component that supports automated operations, i.e. operations that automatically run based on the applications configuration. This type of component may use other general-purpose components but is not used by them. It is started when the application is started and stopped when the application closes.

To easily build and combine components requires that the framework provide:

1. A generic "control panel" that serves as the front end users interface and invokes top level components.
2. A base class definition and "rule set" for general components that supports Windows message distribution, asynchronous component operation and polymorphism.
3. A "manager" and "rule set" for automated operation components.

The second of these requirements corresponds to the central problem that must be overcome in any execution block based framework, COM, CORBA etc. However, unlike these frameworks, this one does not attempt to be language, location or process independent. As a result it is simple and can be understood and used with relatively little effort. Wizards significantly simplify the construction of a project's base file set (i.e. the framework file set) and component "templates" (See wizards.doc for details on the project and component creation wizards.)

3 DESIGN

3.1 Architecture

A control flow perspective of the framework (general components only) is provided in Figure 2. A typical interaction of the framework and general components (ignoring message and mail distribution) using a communication link, is diagramed in Figure 3. The object (inheritance) hierarchy used in the framework is provided in Figure 4. A brief description of the principal parts of the framework follows:

Application - provides the compiler entry point and initializes the applications main window; handles the "About" menu item; provides Windows message distribution services

Control panel - provides the initial human interface to include "launching" top level components via menu interaction; provides necessary administrative functions; provides any inter-locking control in the usage of other components

Mailbox - provides the means for asynchronous communications, completion messages, between components or a top level component and the control panel

Communication - provides facilities to communicate with the device being controlled; the current implementation provides either serial or TCP/IP socket support; a separate class provides the capability to manage multiple communication links.

File manager - provides I/O operations for the file system used by the control application

Child Console - provides text display services on a child window's client area

Application log - provides data/time stamped event logging

Application activity status - provides display of current activity status lines

Progress status - provides independent progress status bar

Operation component base class - provides "wrapper" for a user developed operation component

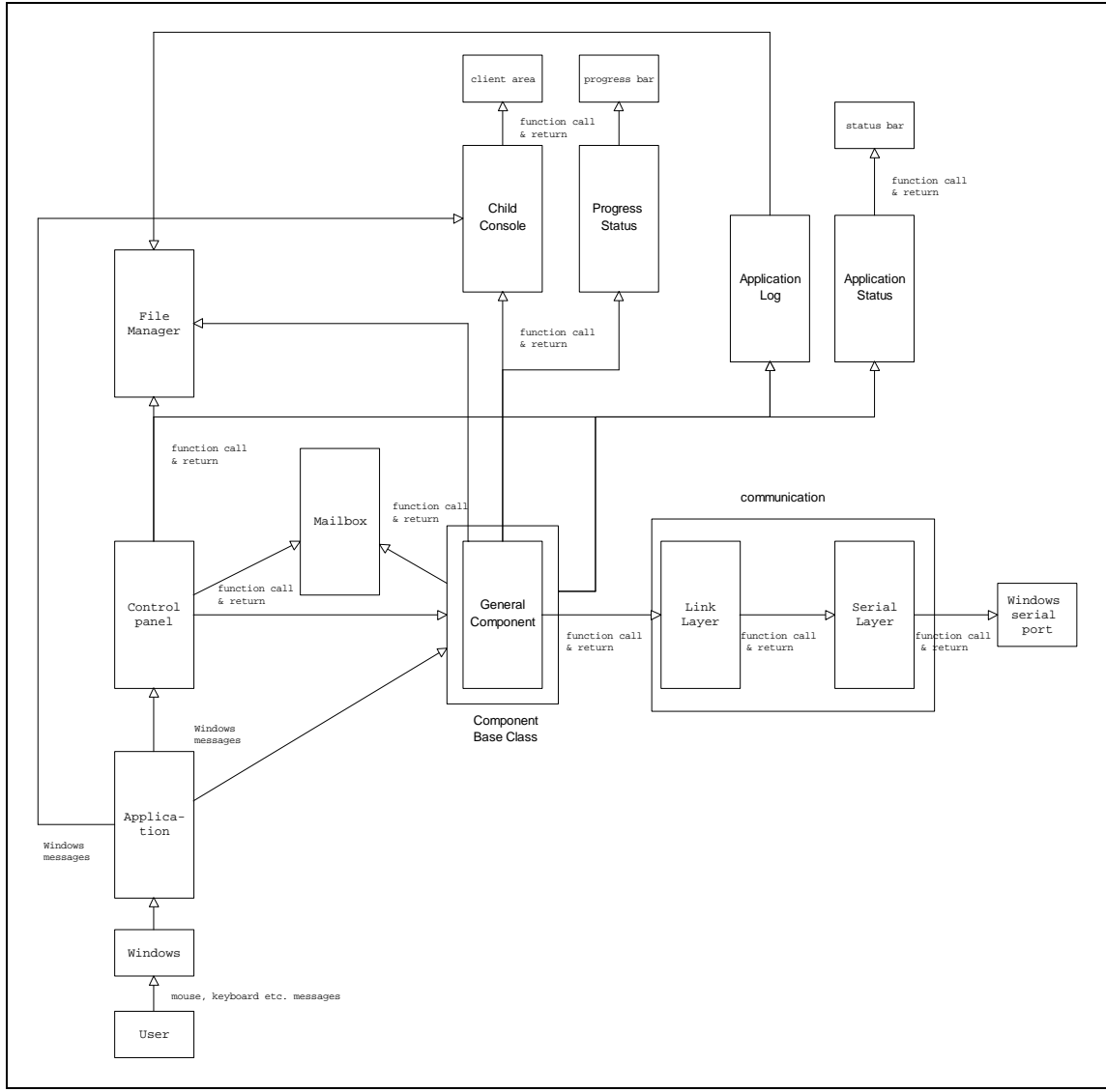


Figure 2, Framework Control Flow

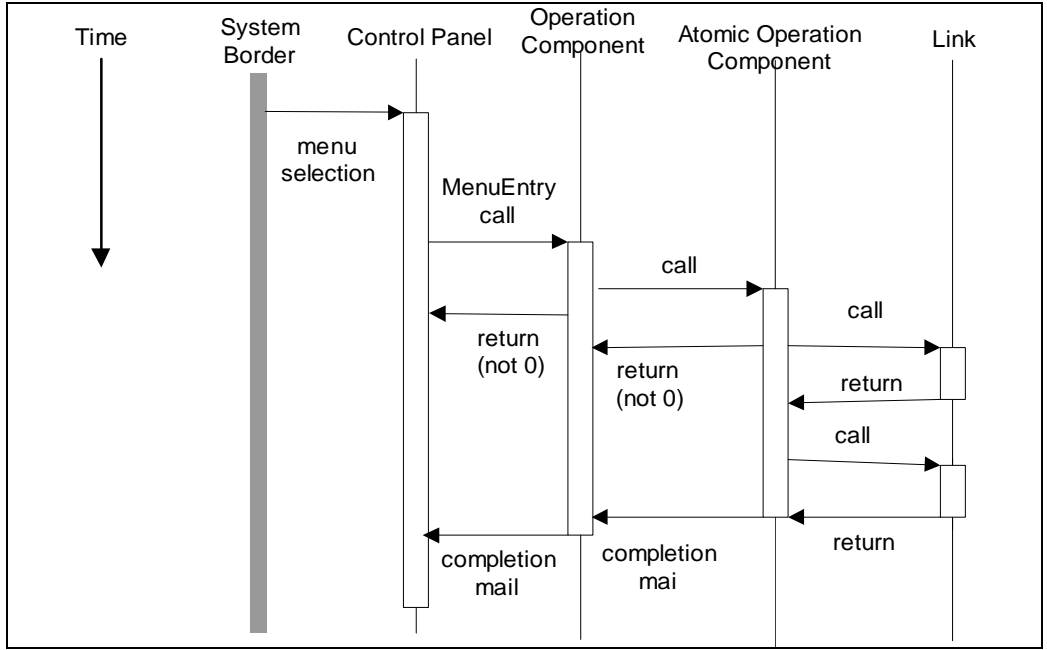


Figure 3, Typical Interaction

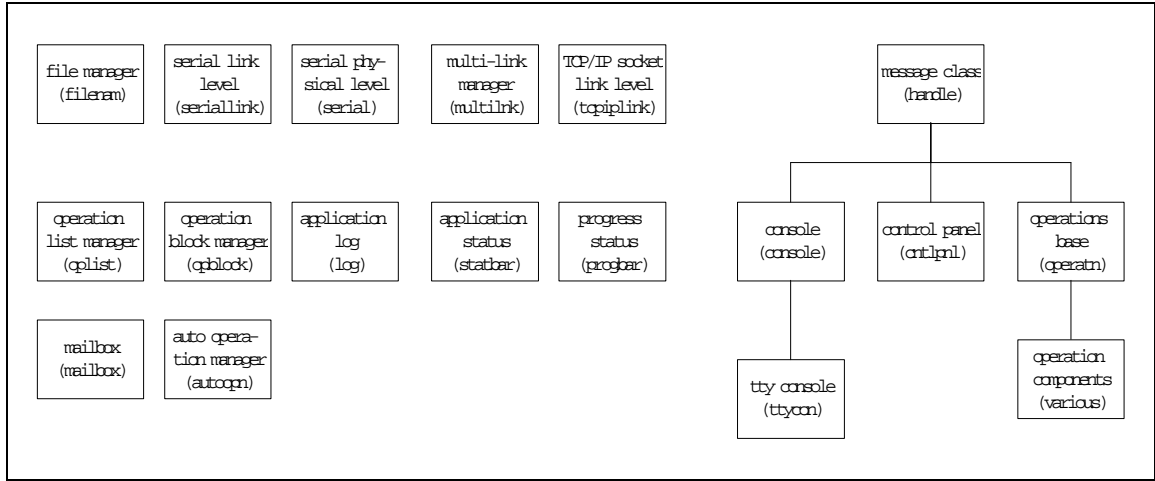


Figure 4, Object Hierarchy

In general a component is composed of two files. It is defined in a header file, *.H, it is implemented in a C++ source file, *.CPP. Its window resources (if any) are defined in a single central project resource script file, *.RC. In a few cases components may share some code with another component using a "neutral" C++ source file. Invocation rules and mailboxes tie general components together. The basic rules for general components are:

1. All component activities are started with a function call.
2. If an activity will require substantial time to perform and can be achieved in an asynchronous manner then a two phase interaction is used. The initial interaction is the return from the function call. It provides the component's ID or zero (an illegal ID number) as a return. The non-zero return indicates that the activity is started and will be completed some time latter. The second or completion interaction is composed of a mail posting to the other component (or control panel) that started it.
3. The component must provide the following minimum public interface:

OPERATION MESSAGE HANDLER

Description: Component's Window's message handler.

Called Parameters: message - message type code
wparam - message word parameter
lparam - message long parameter

Returned value: message handled (TRUE) or not (FALSE)

BOOL component_class::msghandler(UINT, WPARAM, LPARAM)

MENU ENTRY POINT

Description: Provides the menu entry point function defined as virtual in the base operation class definition. This function ALWAYS uses the asynchronous interaction mode.

Passed parameter: none

Returned value: operation started (valid operation ID) or not (0)

unsigned int component_class::MenuEntry(void)

DYNAMICALLY CREATE AN OBJECT

Description: Provides the capability to dynamically create an object of this class. This is declared as a static function.

Passed parameters: activity - display status (TRUE or FALSE)
lptr - pointer to link object to use
id - callers ID

Returned value: pointer to the object

```
void *component_class::Create(BOOL activity,link *lptr,unsigned
int id)
```

DYNAMICALLY DESTROY THE OBJECT

Description: Destroys the object. This is declared as a static function.

Passed parameter: optr - pointer to a connect object to destroy.

Returned value: none

```
void component_class::Destroy(void *optr)
```

4. Components should be "thread safe" if multiple instances can be created. This implies that it will operate correctly if multiple threads, each with their own object instance of that class, operate simultaneously. These can be in general be satisfied by (NOTE: examples of these implementation specifics can be found in the generic control application example):
 - a. If static data structures/variables are used in a component that can have multiple instances, mutual exclusion must be provided by the component.
 - b. If a dialog is used, it is created with the DialogBoxParam call with the LPARAM set to point to the object invoking the dialog. Since only modal dialogs are supported, mutual exclusion at the dialog level is assured; therefore the pointer can be saved within the dialog handler as a static variable.
 - c. If a thread is used, it is created with the passed parameter being a pointer to the object to which the thread "belongs".
 - d. Any events etc. that are not to be shared between objects (instances of a component) should not be named.
5. Any component that is invoked directly from the menu requires an entry in COMPONT.H. This entree is normally created by the component wizard.

A generalized interface and organization diagram for a component is provided below:

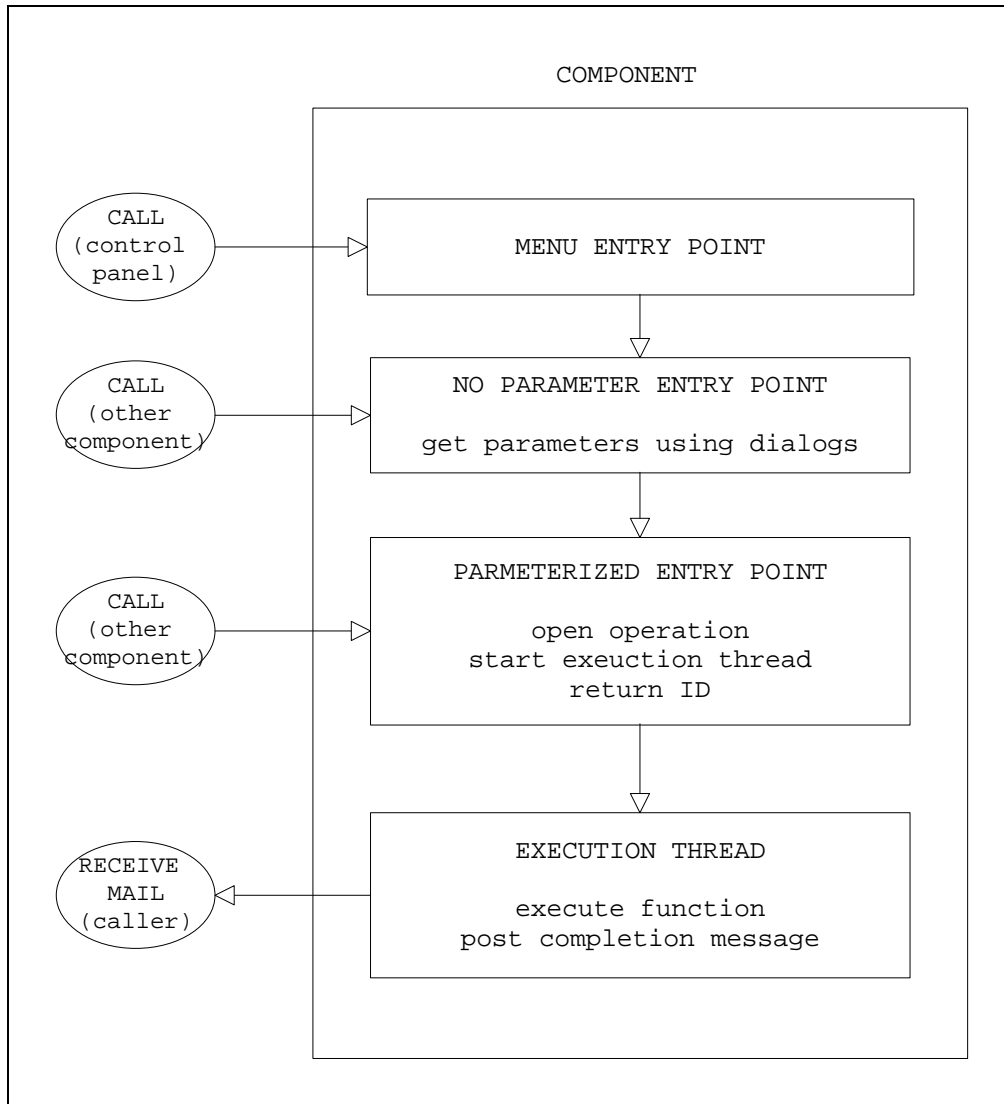


Figure 5, Typical General Component Interface and Organization

Automated operation components are different in function and structure from general components and follow a different set of rules. An automated operation is started when the application opens and is not closed until the application is closed. During that time it periodically "awakens" and does its intended function. This may include the use of general components. In C++ this is accomplished using an execution thread that has a "while forever" loop. The rules for automated operation components are summarized below:

1. Each automated operation component must provide the following static public interface:

OPERATION START ENTRY POINT

Description: Automated operation start entry point. It creates the object and then starts the execution of the free-wheeling thread.

Passed parameters: none

Returned value: none

```
void auto_component_class::Start(void)
```

CLOSE THE COMPONENT

Description: Stops the objects execution thread and destroys the object.

Passed parameter: none.

Returned value: none

```
void autoexchange::Close(void)
```

2. The start function must perform the following actions:
 - a. Check that another instance of the component is not running.
 - b. Create the object and start the free wheeling execution thread.
3. The close function must stop the components executing thread if it is running and delete the object if it exists.
4. The execution thread of an automated component must use the facilities provided by the automatic operation manager to assure that automatic operations will not run while a user invoked component is in progress or another automatic operation is running. The general scheme is pseudo-code is:

```
WHILE (FOREVER)
    calculate time to wait
    Sleep for calculated time
    RunAutoOpn [automatic operation manager]
    logic of component
    AutoOpnComplete [automatic operation manager]
END WHILE
```

5. The component should input and check its configuration parameters prior to start the execution thread.
6. If the operation cannot start the user should be notified.
7. Any automated operation component must have an entree in AUTOCOMPONENT.H. The component wizard normally creates this.

3.2 Detail

The design of each part of the framework is provided in the following sections. The design of the example components that make up the "generic control application" is provided in Appendix A.

3.2.1 Application

Application provides the basic services necessary in a Windows environment to function. These include:

1. The compiler required initial entry point.
2. Initialization of the application's window, using application specific entries in APP.H, and the control panel (using the control panel class).
3. The Windows main thread message loop.
4. The Windows registered message handler for the application window.
5. Handling of the destroy message (WM_DESTROY).

In addition it:

1. Provides the ability to display a sign on banner.
2. Handle the "about" dialog.
3. Creates the globally available key file names with full paths.
4. Provides Windows message distribution functions for other objects.

The last of these is key to the proper operation of the system. It allows any component to register itself and therefore get Windows' messages. This in turn makes it possible to do a component's message handling (if any) in the components class not in a centralized (and completely unwieldy) message handling loop.

The following items are required for correct initialization and/or operation should be uniquely defined in APPL.H for a specific application:

CFG_FILE_NAME - the name of the applications configuration file

APPL_TITLE - application's window caption

MENU - application's menu ID name

ABOUT_DIALOG - about dialog's name

HELP_FILE - application's help file name

SIGNON_BANNER - display sign on banner (TRUE) or not (FALSE)

SIGNON_WAIT - time (in milliseconds) to display the signon banner, ignored if SIGNON_BANNER is FALSE

MULTI_INSTANCE - more then one instance of the application is permitted (TRUE) or not (FALSE)

The prototypes of public functions are also provided in APPL.H. These functions provide the ability to register and unregister a message handler. Any component that wants to use this feature must be defined as a class derived from the "msg_class" class defined in HANDLE.H. This base class really serves no purpose other then the declaration of the virtual function "msghandle". In addition, a means to get key application level handles is provided. A more complete definition of the public functions provided by Application follows:

REGISTER A MESSAGE HANDLER

Description: Provides the means for an object to register itself for receiving messages.

Passed parameters: optr - pointer to the object registering

Returned value: successful (TRUE) or not (FALSE)

mh_entry *RegisterMsgHandler(msg_class *optr)

UNREGISTER A MESSAGE HANDLER

Description: Removes the indicated message handler.

Passed parameter: mhptr - pointer to node to remove

Returned value: none

```
void UnRegisterMsgHandler(mh_entry *mhptr)
```

GET THE APPLICATIONS MAIN WINDOW HANDLE

Description: Returns the handle for the applications main window.

Passed parameters: none

Returned value: applications main window handle

```
HWND GetApplWndHandle(void)
```

GET THE APPLICATIONS HANDLE

Description: Returns the application instance handle.

Passed parameters: none

Returned value: applications instance handle

```
HINSTANCE GetApplHandle(void)
```

Application is implemented in the file APPL.CPP. It is the only part of the framework not implemented as a C++ class. The initialization functions are standard Windows programming and will not be further discussed. (NOTE: In a few instances conditional compile statements have been used to differentiate between Windows 95/NT and CE.) The basic message distribution function is based on two specific items. First, the internal data structure is nothing more than a single linked list of blocks, `mh_entry`, which provide an object pointer. The data structure is defined as:

```
struct mh_entry
{
    mh_entry *next;
    class msg_class *obj_ptr;
};
```

```
mh_entry mhe[NO_MHE];
```

```
mh_entry *reg;
mh_entry *unused;
```

Registered message handlers are found in the `reg` list while

unused blocks are in the unused list. The register and unregister functions just remove and insert blocks.

The second is the requirement that a registered message handler return a Boolean to indicate if the message was handled (TRUE) or not (FALSE). This convention is enforced by the pure virtual message handler function definition in the "msg_class" class definition, see HANDLE.H.

To assure thread safe operation, the public routines, RegisterMsgHandler, UnRegisterMsgHandler, along with the private function that uses the message distribution data structure, DistMsg, are implemented using a critical section. This assures that any changes or uses of the data structure are done indivisibly.

3.2.2 Control Panel

The control panel provides the "user entry point" for the control application framework. It has two basic functions, first to initialize the other parts of the framework used in most or all cases, the communication links, application log and application status bar, and second to provide the first level user interface through its application registered message handler.

The control panel class is defined in CNTLPNL.H. Other than a constructor and destructor its public definition provides the means to initialize and close the control panel and for it to Windows handle messages. The detailed definitions are:

CONTROL PANEL INITIALIZATION

Description: Provides for initialization of a control control panel.

Called parameters: wnd - handle for current window
app - handler for current application
cbwnd - handle for command bar in CE
(defaults as invalid handle)

Return Values: none

```
void cntlpanel::Init(HWND wnd,HANDLE app,HWND cbwnd)
```

CONTROL PANEL'S REGISTER MESSAGE HANDLER

Description: Takes care of handling Window's messages pertinent to a control panel object.

Called Parameters: message - message type code

wparam - message word parameter
lparam - message long parameter

Returned value: message handled (TRUE) or not (FALSE)

BOOL cntlpanel::msghandler(UINT message, WPARAM wparam, LPARAM lparam)

The initialization routine performs the following actions:

1. It initialized the communication link or links using the mutli-link management component. If a communication link is not established the user is altered to this with a warning message.
2. It initilizes the application's log file and status bar.
3. It gets and saves the handle for the system's menu.
4. It initializes its mailbox and starts its free running component reply handler (mail reception) thread.
5. It calls the automated operation manager's entry point to start any automated operations.
6. It registers with the Application for Window's message distribution.

The control panel's menu message handler provides the base level user interface. Specifically it handles messages generated by user menu interactions. Based on menu actions it will launch an operations component to provide the function requested or in some cases use Window's provided facilities to provide the function. Where the component is actually dynamically created (the new operation in C++) the registered message handler takes care of destroying the component when it closes.

In addition, the control panel provides interlock mechanism to prevent improper usage. Currently two interlocks are used:

1. All operations that inherently require a device connection are disabled until you have connected with the embedded system.
2. Communication settings are only enabled when not connected with a device.
3. Once an operation is started, another operation is not allowed

until the operation completes.

The control panel uses facilities in the operation list class and virtual functions defined in the operation base class to dynamically create/destroy and invoke components without needing to know the exact nature (i.e. class) of the specific component (polymorphism). Means to manipulate the defined list of menu invoked components is provided by OPNLIST.H/.CPP (see section 3.2.4) Tracking a functioning component is maintained using the facilities of the operation block linked list class defined in OPBLOCK.H and implemented in OPBLOCK.CPP (see section 3.2.5). The overall operations of the user interface is summarized below:

1. When a user selects a menu item the control panel's registered message handler picks up the Windows' message.
2. The message handler uses facilities in the operation list class to find a component that matches with the menu item's ID.
3. If a match is found the component is dynamically created using facilities in the operation list class and component.
4. The MenuEntry routine in the component is invoked. If the routine returns a legal ID (i.e. the operation is started and final notification of completion etc. will come latter) then:
 - a. Facilities in the operation list class are used to disable all other top-level components and
 - b. Key information about the component, primarily its ID, is kept in a linked list using the facilities of the operation block linked list object.
5. The mail reception thread within the control panel gets any mail posted for the control panel (i.e. completion messages).
6. Incoming mail is checked for a match with a functioning component using the facilities of the operation block linked list object.
7. If a match is found, facilities in the operation list class provide a means to determine the response type.
8. Based on the response type the appropriate actions are taken (e.g. "enabling component operations at the menu level using facilities of the operation base class) and the component's information block removed from the linked list.
9. Almost all of the control panel implementation is menu neutral

in nature. The only exception is the handling of the "Admin" and "Help" menu choices. These may require modification for specific applications.

3.2.3 Operation Base Class

Operation components provide the core of the control applications functionality. They allow the control application (and the user through it) to configure, program and/or control an embedded system.

To eliminate redundant code, the more complex components are expected to use the basic components to do much of its work. Because of this it is extremely important that they be created in a manner that follows all the framework rules. In addition care must be taken to make sure that these components operate properly when created in either a static or dynamic nature. Examples of operation components are provided in the generic control application example. Normally a new component is created using the Component Wizard. This is documented in wizard.doc.

Operations typically require the use of framework classes to fulfill their jobs. In general, access to the most important of these, a communication link to the device, must be provided for any operation to work. Normally operations either directly use this component to communicate with the embedded system or invoke basic operations that do (or in some case both). This means that a component is inherently asynchronous. If the component's operation is started it lets its invoker know that it has completed and the status of the completion by posting mail for the invoker. Where data needs to be transmitted there is a unique data block provided in the component. The location of this block is part of the completion mail.

Access to the console component is useful for displaying the state of the operation etc. but is not mandatory for use of a component. This is especially true when a basic operation component is being invoked by a more complex operation component and the wish is to eliminate possible confusing aggregations of console messages. Therefore a operation component can not assume that it has access to the console and must check before trying to use it.

The operation base class defined in OPERATN.H provides the aspects shared by all components. This includes the use of the "msg_class" class as a base and key variables/data structures. Two variables, my_id and caller_id, provide the means to assure correct posting and receipt of completion mail. The variable caller_id is set by the base classes constructor. It is the ID of

the component that created this component (i.e. the invoker of the operation). The variable `my_id` should be set using the value returned from the `OpenOpn` call (see below). This variable must be used for the initial return to a `MenuEntry` invocation if the operation is started. Mail responses from other components is provided in `rsp`:

```
struct asynch_response rsp;

struct asynch_response
{
    enum opn_msg_wparam wparam;
    void *response;
};
```

where the fields are defined as:

`wparam` - status parameter (succeed or fail) unique to the component

`response` - pointer to the components information block or `NULL`

In addition the base class provides a number of support functions that eliminate code redundancy and make it easier to construct a new operation component. A detailed definition of the function interface follows:

ACTIVITY MESSAGING

Description: Displays the message on the application status bar and makes a log entry if activity status was `TRUE` (i.e. not silent). Always logs item with `must_log` set to `TRUE`.

Passed parameter: `msgptr` - pointer to string to output
`must_log` - must log event if silent

Returned value: none

```
void opn::Message(TCHAR *msgptr,BOOL must_log)
```

OPEN AN OPERATION

Description: Takes care of registering an operation for message distribution, constructing a mailbox for use with other components and provides the component its ID.

NOTE: The returned ID is generated by the `InitMailBox`

Passed parameter: none

Returned value: if successful this returns a number other than zero, the number is the component's ID and should be stored in my_id and used in the creation of any other operation component

unsigned int opn::OpenOpn(void)

CLOSE AN OPERATION

Description: Takes care of closing the operation. This includes unregistering the operation for windows message distribution, closing the components mail box, displaying any final message, if a console is available, and posting the completion mail to the invoker

Passed parameters: msg - message to display
type - wparam for the completion message
data - pointer to data block (if any)
associated with the response

Returned value: none

void opn::CloseOpn(char *msg,opn_msg_type type,void *data)

CLOSE AN OPERATION

Description: Takes care of closing the operation. This includes unregistering the operation for message distribution, closing the components mail box and displaying any final message if appropriate.

Passed parameters: msg - message to display

Returned value: none

void opn::CloseOpn(char *msg)

WAIT FOR THE ASYNCH RESPONSE FROM ANOTHER COMPONENT

Description: Provides the means to wait for an asynchronous response from another component. This call blocks the calling thread until the response is received. The actual response is saved in the variable rsp.

Passed parameter: sender - id of component waiting for

Returned value: none

```
void opn::WaitOpnResponse(unsigned int sender)
```

To assure proper operation in a mutli-threaded environment, each of the public functions, Message, OpenOpn and CloseOpn, are protected using critical sections (one for Message and another for OpenOpn and CloseOpn).

Beside the above support, the operation base class defines some pure virtual functions that a component must implement. These are defined below:

BOOL msghandler(UINT,WPARAM,LPARAM) - the registered Window's message handler

unsigned int MenuEntry(void) - the component's menu invoked entry point (this is used by the control panel)

3.2.4 Component List Class

The component list class provides a convenient means to dynamically manipulate menu-invoked components (this feature is primarily intended for use by the control panel). This is based on an array of operation data structures that include entry points for static functions. The operation data structure and the array are defined as:

```
struct operation
{
    TCHAR *title;
    unsigned int menu_id;
    void *(*Create)(BOOL,LNK *,unsigned long);
    void *(*Destroy) (void *);
    BOOL repeat;
    BOOL always_active;
}
```

```
static struct operation op[];
```

A brief explanation of each field follows:

title - a title string

menu_id - the Windows ID for the menu item associated with the operation

Create - pointer to the routine that dynamically creates an object (a static function in the component).

Destroy - pointer to the routine that deletes a dynamically created object (a static function in the component).

repeat - indicates that the operation should be repeated, automatically re-run after completion, (TRUE) or not (FALSE).

always_active - indicates that the operation can be invoked event when there is not connection, (TRUE) or not (FALSE).

"op" entriees are normally built by the component wizard in the file COMPONENT.H. The implementation of an object to manipulate the components in op is provided in OPNLIST.H/.CPP. The following functions are provided to operate on the array:

FIND OPERATION THAT MATCHES A MENU ID

Description: Finds the operation that matches a menu ID, if any.

Passed parameter: menu_id - the menu ID to match

Returned value: the matching operations handle or NO_OPERATION

OPN_HNDL opn::MatchMenuId(WPARAM menu_id)

FIND OPERATION THAT MATCHES A OPERATION MESSAGE COMPLETION RESPONSE

Description: Finds the operation that matches one of three possable operation message response parameters (success or fail).

Passed parameter: response - response parameters to match

Returned value: the matching operations handle or
NO_OPERATION

OPN_HNDL opn::MatchResponse(WPARAM response)

CHECK OPERATION MESSAGE COMPLETION RESPONSE FOR TYPE RESPONSE

Description: Returns the response type for an operation message.

Passed parameters: operation - the operation's handle
response - the response

Returned value: the match type or not operation response

enum opn_msg_type opn::ResponseType(OPN_HNDL operation,WPARAM response)

RETURN THE TITLE FOR THE INDICATED OPERATION

Description: Returns a pointer to the title, a null terminated string, of the indicated operation.

Passed parameters: operation - the operation's handle

Returned value: pointer to the operations title or NULL

```
char * opn::Title(OPN_HNDL operation)
```

CREATE THE INDICATED OPERATION COMPONENT

Description: Invokes the Create function of the indicated component.

Passed parameters: operation - the operation's handle
appl - application handle
wnd - window handle
console - console present or not
lnk - pointer to serial link
id - callers id

Returned value: pointer to created component or NULL

```
void *opn::CreateOpn(OPN_HNDL operation,HANDLE appl,HWND wnd,BOOL console,link *lnk,unsigned int id)
```

DESTROY THE INDICATED OPERATION COMPONENT

Description: Invokes the Destroy function of the indicated component.

Passed parameter: operation - the operation's handle
opptr - pointer to the object

Returned value: component destroyed (TRUE) or not (FALSE)

```
BOOL opn::DestroyOpn(OPN_HNDL operation,void *opptr)
```

RETURN REPEAT STATUS OF INDICATED OPERATION

Description: Returns the repeat status of the indicated operation.

Passed parameter: operation - operation's handle

Returned value: repeat status (TRUE or FALSE)

BOOL opn::Repeat(OPN_HNDL operation)

ENABLE OR DISABLE ALL THE OPERATION'S MENU ITEMS

Description: Enables or disables the entire operation menu.

Passed parameter: action - enable (TRUE) or disable (FALSE)
menu - menu handle

Returned value: the menu ID or zero.

void opn::AllOpnMenu(BOOL action,HMENU menu)

ENABLE OR DISABLE THE CONNECT MENU ITEM

Description: Enables or disables the "Connect" operations menu item.

Passed parameter: action - enable (TRUE) or disable (FALSE)
menu - menu handle

Returned value: the menu ID or zero

void opn::ConnectMenu(BOOL action,HMENU menu)

ENABLE OR DISABLE THE DISCONNECT MENU ITEM */

Description: Enables or disables the "Disconnect" operations menu item.

Passed parameter: action - enable (TRUE) or disable (FALSE)
menu - handle to menu

Returned value: the menu ID or zero

void opn::DisconnectMenu(BOOL action,HMENU menu)

ENABLE ALL THE MENU ITEMS THAT ARE ALWAYS ACTIVE*/

Description: Enables operations menu item that are always active.

Passed parameter: menu - handle to menu

Returned value: none

void oplist::EnableConstantMenuItems(HMENU menu)

ALWAYS ACTIVE ITEM

Description: Determines if an item is always active.

Passed parameter: operation array handle

Returned value: always active operation menu item (TRUE) or
not (FALSE)

BOOL opnlist::AlwaysActive(OPN_HNDL ohndl)

3.2.5 Operation Block Linked List

The control panel to maintain a list of components that are currently executing uses this framework class. The class is defined in OPBLOCK.H and implemented in OPBLOCK.CPP. The class does nothing more than keep a single linked list of component data where this is defined as:

```
struct opn_block
{
    struct opn_block *next;
    class opn *opptr;
    unsigned int op_id;
    unsigned short int ohndl;
}
```

where the fields are defined as:

next - pointer to next block in the list or EOL

opptr - pointer to the object that implements the component

op_id - the component's ID

ohndl - handle for the block

The public functions supported are:

INSERT A NEW BLOCK

Description: Creates and inserts a new block.

Passed Parameter: id - op_id parameter
ohndl - ohndl parameter
optr - opptr parameter

Returned value: block created and inserted (TRUE) or not
(FALSE)

```
BOOL opn_block_sll::Insert(unsigned int id,unsigned short int
ohndl,class opn *optr)
```

FIND A MATCHING BLOCK

Description: Finds a block with a matching op_id, if it exists.

Passed parameters: id - op_id parameter to match

Returned value: pointer to matching block or NULL

```
struct opn_block *opn_block_sll::Find(unsigned int id)
```

REMOVE THE SPECIFIED BLOCK

Description: Removes and destroys the indicated block if it exists.

Passed parameter: bptr - pointer to block to eliminate

Returned value: block removed (TRUE) or not (FALSE)

```
BOOL opn_block_sll::Remove(struct opn_block *bptr)
```

3.2.6 Automated Operation Manager

The automated operations manager provides the means to start and stop all automatic components. In addition it provides facilities required by automated components for correct operation. It is defined in AUTOOPN.H and implemented in AUTOOPN.CPP. Starting and stopping automatic components is based on an array of automatic operation data structures that include entry points defined as static functions. The data structure and the array are defined as:

```
struct auto_operation
{
    TCHAR *title;
    void (*Start)(void);
    void (*Close)(void);
};
```

```
static struct auto_operation autoop[];
```

A brief explanation of each field follows:

title - a title string

Start - pointer to the routine that starts the automatic component (a static function in the component).

Close - pointer to the routine that stops an automatic component (a static function in the component).

The functions used by the control panel to start and stop automated operations is:

INITIALIZE AUTOMATED OPERATIONS

Description: Initialized all the automated operations.

Passed parameters: none

Returned value: none

void autoopns::Init(void)

CLOSE AUOTMATED OPERATIONS

Description: Closes all the automated operations.

Passed parameters: none

Returned value: none

void autoopns::Close(void)

In addition the manager provides two key functions to the automated operations themselves that:

1. Guarantee that only one operation runs at a time.
2. Manual operations and automated operations are mutually exclusive.
3. Once an automated operation starts that it will run to completion unless the hardware is powered down or the process aborted.
4. Minimal feedback is provided so an operator knows an automated operation is running.

RUN AN AUTOMATED OPERATION

Description: Takes care of group actions to let an automated operation run.

NOTE: The returned ID is generated by InitMailBox

Passed parameter: msg - operation's start message

Returned value: the components (mail) ID during the run (i.e. from RunAutoOpn to AutoOpnComplete)

```
unsigned int autoopns::RunAutoOpn(TCHAR *msg)
```

AUTOMATED OPERATION COMPLETED

Description: Takes care of group actions necessary when an automated operation is done.

Passed parameters: msg - operation's completion message

Returned value: none

```
void autoopns::AutoOpnComplete(TCHAR *msg)
```

The proper use of these functions is illustrated below:

```
OperationExecThread(void *param)
```

```
WHILE (FOREVER)
    calculate time to wait to start operation
    IF (wait is longer then 0)
        Sleep
    END IF
    RunAutoOpn(TEXT("Running operation name"));
    logic to implment the automated operation including any
    failure recovery etc.
    AutoOpnComplete(TEXT("Operation name completed"));
END WHILE
```

3.2.7 Mail Box

This is a framework class that provides the basic mechanism for the implementation of the asynchronous interaction model for components. [NOTE: It can in fact be used to support any synchronization and communication requirements in the Win32 threaded environment.] It is defined in MAILBOX.H and implemented in MAILBOX.CPP. Instances will be statically created objects within the control panel, operation base class and automatic operation base class. The class definition is:

Internal data store:

```
static struct mailbox *postoffice
```

```
static HANDLE mail_mutex
unsigned char my_id
```

postoffice - the head node for a linked list of "mailboxes" one for each robot/device task.

mail_mutex - object used to enforce mutual exclusion within the mailbox functions

my_id - the identification of the task associated with the mailbox

A "mailbox" is the data structure that provides the means to send and receive mail. There is one for each task. The structure is defined as:

```
struct mailbx
{
    struct mailbx *next;
    unsigned char id;
    struct mail *msg;
    unsigned char wait_sender;
    unsigned char sender;
    char *message;
    unsigned char size;
    HANDLE receive_wait;
}
```

"Mail" is a data structure used to temporarily hold a message until its receiver picks it up. A mailbox keeps a linked list of waiting mail, in receipt order, using the head node msg. The sender, message and size variables are used to receive mail while waiting. The mail structure is defined as:

```
struct mail
{
    struct mail *next;
    unsigned char sender;
    char *message;
    unsigned char size;
}
```

The task IDs are nothing more than a component's ID.

Public function definition:

INITIALIZE A TASKS MAILBOX

Description: Takes care of creating a task's mailbox etc.

NOTE: There is the potential for a race condition in this routine when "initialized" is false. Therefore, this routine should only be invoked by the same thread or otherwise provided mutual exclusion external to its normal mechanism, mail_mutex.

Passed parameters: controlpanel - the caller is the control panel (TRUE) or not (FALSE)

Returned value: the component's (mail) ID

unsigned int mailbox::InitMailBox(BOOL controlpanel)

CLOSE A MAILBOX

Description: Closes a mailbox if it exists. If this is the last mailbox in the postoffice it also "closes" the postoffice.

Passed parameter: none

Returned value: none

void mailbox::CloseMailBox(void)

POST MAIL

Description: Post a message to another task's mailbox

Passed parameters: recv_id - mail's reciever(s)
message - pointer to mail's message
msg_size - size of message in bytes

Returned value: mail posted (TRUE) or not (FALSE)

BOOL mailbox::PostMail(unsigned char recv_id, char *message, unsigned char msg_size)

WAIT FOR MAIL

Description: Waits for a message to be posted by the indicated sender(s)

Passed parameters: send_id - id of sender(s) to receive mail from
buffer - pointer to message buffer

Returned value: id of sender (or zero if failure)

unsigned char mailbox::WaitMail(unsigned char send_id, char

*buffer)

CHECK FOR MAIL

Description: Checks if mail has been posted by the indicated sender(s)

Passed parameters: send_id - id of sender(s) to check for mail from

Returned value: mail found (TRUE) or not (FALSE)

BOOL mailbox::CheckMail(unsigned char send_id)

Implementation of principal routines:

BOOL InitMailBox(unsigned char id)

WaitForSingleObject mail_mutex

IF (mailbox does not exist for id) THEN

 IF (dynamcally create mailbox) THEN

 add mailbox to end of postoffice list

 CreateSemaphore receive_wait

 set mailbox and my_id to id

 set mail list to empty

 set current to empty

 set return to TRUE

 ELSE

 set return to FALSE

 ENDIF

ELSE

 set retrun to FALSE

ENDIF

ReleaseMutex mail_mutex

BOOL PostMail(unsigned char recv_id, char *message, unsigned char msg_size)

WaitForSingleObject mail_mutex

IF (receiver's mail box exist) THEN

 IF (mailbox owner waiting for mail from sender) THEN

 copy message into buffer

 set mailbox sender and size

 ReleaseSemaphore receive_wait

 set return to TRUE

 ELSE

```

        IF (dynamically create mail) THEN
            set mail's sender to my_id
            copy message
            set mail's size to msg_size
            set return to TRUE
        ELSE
            set return to FALSE
        ENDIF
    ENDIF
ELSE
    set return to FALSE
ENDIF
ReleaseMutex mail_mutex

```

```

unsigned char WaitMail(unsigned char send_id, char *buffer)

```

```

WaitForSingleObject mail_mutex
IF (mailbox exit for my_id) THEN
    IF (mail from send_id in list) THEN
        copy message into buffer
        set mailbox sender and size
        set return to sender
    ELSE
        set wait_sender to send_id
        copy buffer pointer
        WaitForSingleObject receive_wait
        copy message into buffer
        set mailbox sender and size
        set return to sender
    ENDIF
ELSE
    set retrun to 0
ENDIF
ReleaseMutex mail_mutex

```

3.2.8 Device Communications

The ability to establish communication with an embedded system (and therefore configure/program/control it) is supported for three kinds of links: a high efficiency and reliability point-to-point serial link, a TCP/IP socket link and a proprietary protocol. The proprietary link is just a "template" for a user provided protocol and will not be described. A separate class provides the means to manage multiple communication links.

3.2.8.1 Multiple Communication Link Manager

The multi-link class provides routines to manage multiple open serial links. The class is defined in MULTILNK.H. Detailed

definitions for each public function follow:

INITIALIZE THE LINK ARRAY

Description: Initializes the communication links array using the configuration file and dynamically created link objects. This operation takes "possession" of each port defined in the configuration file.

Passed parameters: none

Returned value: number of ports opened (0 to 4)

```
unsigned char multi_link::InitLinks(void)
```

CLOSE ANY OPEN LINKS

Description: Closes any link that is open.

Passed parameter: none

Returned value: none

```
void multi_link::CloseLinks(void)
```

GET LINK AVAILABILITY

Description: Allows the caller to get the availability of indicated communication link.

Passed parameters: name - pointer to link's name string

Returned value: link is available (TRUE) or not (FALSE)

```
BOOL multi_link::LinkAvailable(TCHAR *name)
```

GET A LINK'S POINTER

Description: Allows the caller to get the pointer to a link object.

Passed parameters: name - pointer to link's name string

Returned value: pointer to link object or NULL

```
class serial_link *multi_link::GetLinkPtr(TCHAR *name)
```

SELECT A LINK

Description: Provides means for a user to select a link.

Passed parameters: name - pointer to buffer to receive link's
name string (or NULL)

Returned value: link selected (1- x) or 0

```
unsigned char multi_link::SelectLink(TCHAR *name)
```

The class is implemented in MULTILNK.CPP. The functions are no more than operations on a link list of communication link definition blocs:

```
struct comm_link_bloc  
{  
    struct comm_link_bloc *next;  
    TCHAR name[MAX_TITLE_LEN + 1];  
    class link *lnk_ptr;  
};
```

The class "link" is a virtual class that provides the definition of a generalized communication interface. It is defined in COMLINK.H. The interface is based on the following assumed communication usage model:

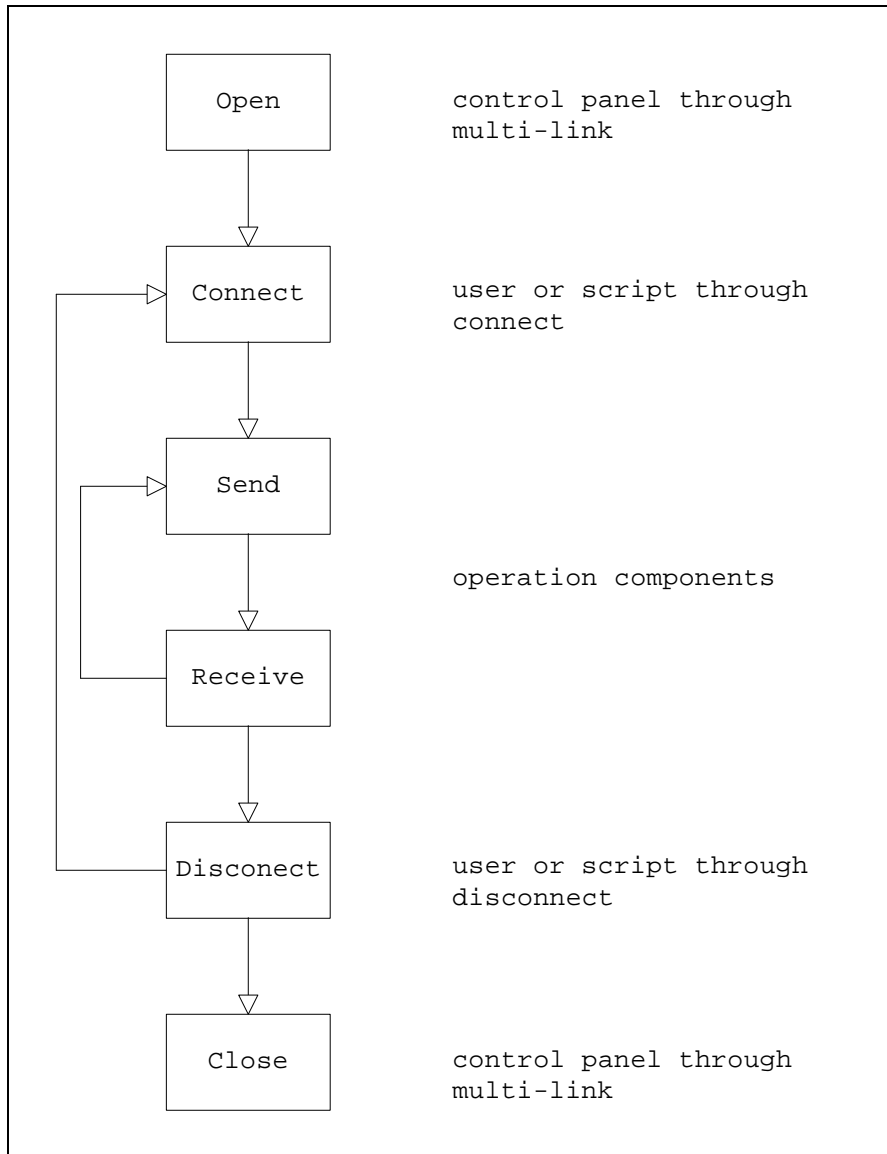


Figure 6, Communication Usage Model

A generic virtual interface that supports this model is provided in COMLINK.H. There are currently two implemented versions:

1. A point-to-point serial link based on Window's serial functions and TAPI.
2. A TCP/IP socket link based on Window's WINSOCK and RAS/DUN.

3.2.8.2 Serial Communication Link

The serial communication components were built as separate classes to maximize reuse and to keep the actual modules that

implement them maintainable. The separation pretty much follows "standard" layering found in most communication protocol implementations. The link level component provides a high reliability packet interface to the other components in the system. The serial port level component provides an interface to the Window's serial port API.

The link level provides packetization, reliability and encoding facilities. The current implementation provides thread based locking protection. If using mutli-threaded access to a single link, then the application must:

1. Use the Lock - Unlock primitives or
2. include its own protection (mutex, semaphore etc.) or
3. expand the protocol to support multi-point to point communication.

The link protocol was designed and implemented to have the following characteristics:

1. Very low complexity so it can be easily implemented on an embedded device (NOTE: an implementation in C is provided in the DOS device simulator).
2. High reliability
3. Good bi-directional binary data transfer rate under normal conditions.
4. Supports a master - slave relationship between the control application (master) and the controller (slave). This assumes that "indivisible" Send - Receive calls are used as in a message based conversation between a single "source" in the control application and a single "sink" on the device.
5. Supports point-to-point physical connection. This includes both local and remote (point to point modem) connections.

The key aspects of the protocol are summarized below:

1. This is a link level protocol which operates with no knowledge of its contents other than whether a packet is the end of a "message" or not. It is assumed that a conversation, the exchange of one or more packets to perform a function such as a file transfer, based context exists above this communication layer.
2. It uses prefix encoding to provide full binary transparency

along with unique control symbols. The prefix used is the ASCII escape character (0x1B). Four items are currently encoded:

start of packet:	<ESC><STX>	
ACK:	<ESC><ACK>	
NAK:	<ESC><NAK>	
escape:	<ESC><ESC>	[not a control symbol]

3. Data being transfer is packetized to provide reliable transport. A packet is defined as:

start of packet
sequence number
data length
data bytes
CRC

sequence number - a byte value, the lowest seven bits are incremented in order, a sequence number is "new" when a new connection is started., the high bit is set if this is the last packet in a message

data length - a byte value, provides the length of the data section, currently limited to 128 bytes less packet/encoding overhead

CRC - two byte CCITT recommended CRC

4. The operation of the protocol is half-duplex with a single packet allowed to be outstanding. A positive, ACK, or negative acknowledgment, NAK, is required for each packet sent. The response is based on the correct sequence and CRC and does not imply that any operation other than receipt has take place. No flow control is provided.
5. If a positive or negative acknowledgment is not received within 30 msec then the packet is re-transmitted.
6. A failed packet, either NAKed or timed out, will be sent a total of five times before the protocol will fail.
7. No flow control is used.

The protocol can be described as two finite state machines. The receive machine is provide in Figure 7 and the send machine in Figure 8.

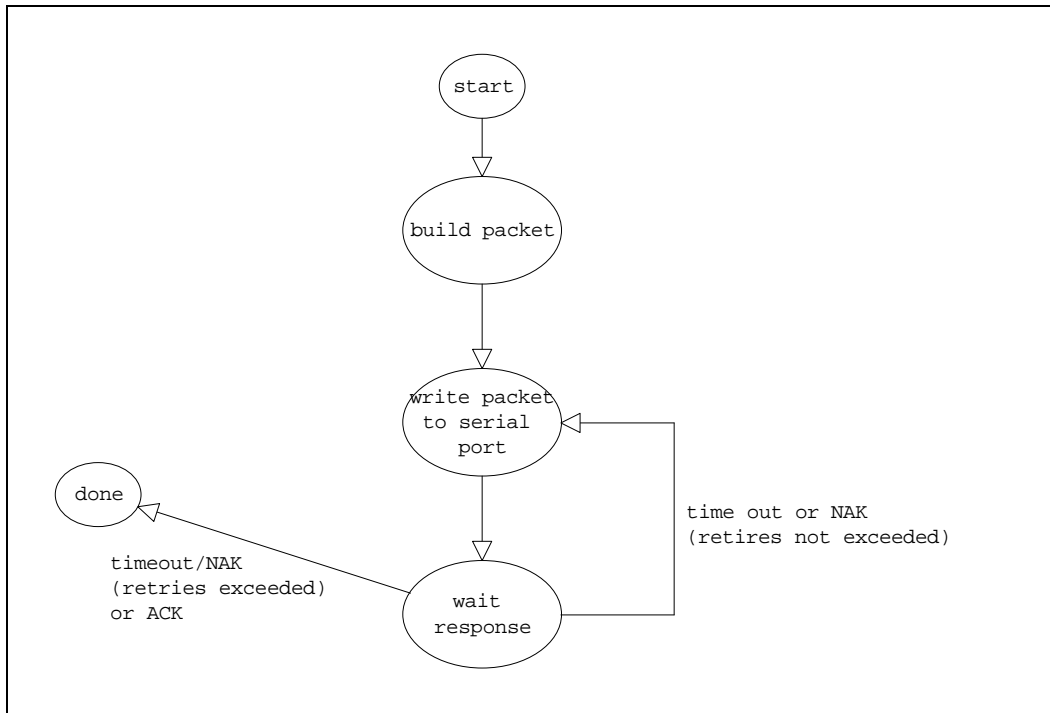


Figure 8, Send State Diagram

The link layer public functions are defined in LINK.H. The send and receive functions are blocking in nature. Since the PC is always the master, a successful send should always proceed a receive. A detailed definition of the public functions follows:

OPEN THE LINK

Description: Opens the indicated port.

Passed parameters: port - comm port to use 1 to 3
win - window handle

Returned value: open ok (TRUE) or not (FALSE)

BOOL serial_link::Open(char port,HWND win)

CLOSE THE LINK

Description: Provides a means to close the link without destroying the object.

Passed parameters; none

Returned value: none

void serial_link::Close(void)

FLUSH THE LINK

Description: Flushes the link and its serial port by rebuilding the connection.

Passed parameter: none

Returned value: flush was successful (TRUE) or not (FALSE)

BOOL serial_link::Flush(void)

SEND A MESSAGE

Description: Sends the indicated message as a packet. This is a blocking call. Successful completion means that the message was transmitted and acknowledged.

Passed parameter: length - size of message to send
buff - pointer to message
end - last packet (TRUE) or not (FALSE)

Returned value: transmission was (TRUE) or not (FALSE)

BOOL serial_link::Send(unsigned char length,unsigned char *buff,BOOL end)

RECEIVE MESSAGE

Description: Waits for the receipt of a message. This is a blocking call. Successful completion means that a message of the indicated length is in the provided receipt buffer.

Passed parameter: rbuff - pointer to receive buffer (must be large enough to hold a maximum sized individual message)
lptr - pointer to variable to store the length of the message received
timeout - number of milliseconds to wait for the first character before timing out, a value of 0 indicated indefinite, must be larger then 2500 msec (this prevents timeout during worst case reception conditions)

Returned value: went into receive mode (TRUE) or not (FALSE)

BOOL serial_link::Receive(unsigned char *rbuff,unsigned short int *lptr, unsigned int timeout)

appropriate state
flushes -
upload_stats
no_pkts - number of packets transmitted
re_trans - number of packet re-transmission
ack_timeouts - number of times the retry count was exceeded on an attempted packet transmission

PARAMETER SETTING DIALOG

Description: Provides the GUI means to set the links parameters.
Currently this is limited to setting the serial port parameters.

Passed parameter: hndl - application handle

Returned value: none

void link::Dialog(HANDLE hndl)

CONNECT

Description: Implements the communications link level connect.
The current implementation supports a point-to-point "hard wired" serial connection (where a successful open is a connect) or a remote modem connection.

Passed parameters: none

Returned value: successful (TRUE) or not (FALSE)

BOOL serial_link::Connect(void)

DISCONNECT

Description: Implements severing the communications link level connection. The current supports a point-to-point "hard wired" serial connection (which always returns true) or a modem based connection.

Passed parameters: delay - time to wait in msec for other side to disconnect
0 if disconnect without wait

Returned value: none

The link layer is implemented in LINK.CPP. Most of the detail design directly derives from the protocol definition and the public interface definition above. The only real exception to

this being the overall design of a practical receiver and the connect/disconnect functions. The receiver needs to handle the reception of device responses to transmissions and incoming packets in a manner that will not malfunction in the face of obvious failures such as dropped acknowledgments. This is best achieved with a separate "freewheeling" thread that can synchronize the send and receive functions. In pseudo-code the receiver looks like:

```
WHILE (forever)
  wait for serial character available
  Read character(s) into input buffer
  FOR (each character)
    decode input
    CASE ENTRY (symbol)
      CASE SOP detected
        IF (not waiting for SOP) THEN
          increment bad state counter
        ENDIF
        start packet reception
      ENDCASE

      CASE ACK detected
        set last response to ACK
        signal response received
      ENDCASE

      CASE NAK detected
        set last response to NAK
        signal response received
      ENDCASE

      CASE no symbol
      ENDCASE

      OTHERWISE
        IF (not waiting SOP) THEN
          pass to packet reception
        ENDIF
      ENDCASE
    END CASE ENTRY
  ENDFOR
ENDWHILE
```

Hard-wired serial links require nothing at the link level for connection/disconnection (but may at the message level). With modems, however, this is not the case. The current implementation uses TAPI version 1.5. [Note: This version is

supported under all the Win32 environments.] It is based on the following assumptions:

1. There is only one modem line and it exists on the indicated port or modems have unique names and this name is provided in the configure file (see the configuration file section).
2. There may be other active TAPI definitions other than 1 (PC to PC connections etc.). Therefore the TAPI "line" must be checked for having a modem (you can not assume that the first thing the TAPI gives you is the modem line you want.)
3. A simple text file list of names and phone number in dialable format (i.e. numbers only) is provided for user selection of site to connect to.

The use of TAPI version 1.5 is covered in depth in Communications for Programming in Windows 95 from the Microsoft Press (and other places) and will not be covered here. There are a few key things that are not well documented and need to be noted:

1. Under TAPI version 1.5 or earlier only the call back method is supported. The call back method causes an invisible window to be created for the thread that calls lineInitialize. This invisible window takes care of getting the call back messages from the TAPI service to the applications call back routine. Therefore the initializing thread must provide its own message loop to get asynchronous TAPI returns.
2. If the COM port on which the TAPI line resides is open, the call to lineMakeCall will fail (though the line calls prior to that work fine). Therefore, the single modem or unique known name modem assumption.

The serial port class provides a generic relatively friendly C++ wrapper interface to the Windows serial communication API. It operates in a non-overlapped manner when working with a serial port. When working with a TAPI port, it may be overlapped or non-overlapped depending on the TAPI's underlying implementation. This requires that the Write and Read routines (see below) must be implemented in a manner that works correctly in either case. Because of the assumed general nature of supported communication (packetized protocols) the interface for sending and receiving is based on the idea of a complete "packet buffer" rather than individual characters.

The class is defined in SERIAL.H. Detailed definitions for each public available function follow:

INITIALIZE THE PORT

Description: Initializes the serial communication port. Note the work around used to get baud rates not supported by the BuildCommDcb function.

Passed parameters: wnd - window handle
port - serial port id (COM1, COM2, COM3 or COM4)
baud - any Window's supported baud rate
parity - parity set for character (NONE, ODD, EVEN)
bits - number of bits per character (7 or 8)
stop - number of stop bits per character (1 or 2)

Returned values: initialization successful (TRUE) or not (FALSE)

```
BOOL serial::Open(HWND wnd,SCHNL port,int baud,PARITY parity,char bits,char stop)
```

CLOSE THE PORT

Description: Closes the port if it is open.

Called parameters: none

Returned values: none

```
void serial::Close(void)
```

READ ONE OR MORE CHARACTERS FROM THE SERIAL PORT

Description: Supports reading from the serial port. This includes error condition clearing.

Passed parameter: cpnt - pointer to a character buffer
max - maximum number of bytes to read
err - pointer to error counter

Returned value: number of characters read

```
int serial::Read(char *cpnt,int max,int *err)
```

WRITE ONE OR MORE CHARACTERS TO THE SERIAL PORT

Description: Supports writing to the serial port.

Passed parameter: cpnt - pointer to a character buffer

no - number of bytes to write

Returned value: number of characters written

int serial::Write(char *cpnt,int no)

FLUSH THE INPUT BUFFER

Description: Empties the Window's level input buffer

Passed parameters: none

Returned value: input buffer flushed (TRUE) or not (FALSE)

BOOL serial::FlushInputBuffer(void)

WAIT FOR A RECEIVED CHARACTER

Description: Waits for a character to be received or a timeout.

Passed parameters: timeout - time in milliseconds to wait for a character, value of zero means wait forever

Returned value: character received (TRUE) or not (FALSE)

BOOL serial::WaitChar(unsigned int timeout)

GET THE PORTS SETTINGS

Description: Provides the serial communication ports current settings.

Passed parameters: port - pointer to a serial port id (COM1, COM2 or COM3)
baud - pointer to a baud rate (1200, 2400, 4800, 9600, 14400, 19200 or 28800)
parity - pointer to a parity set for character (NONE, ODD, EVEN)
bits - pointer to the number of bits per character (7 or 8)
stop - pointer to number of stop bits per character (1 or 2)

Returned values: information available (TRUE) or not (FALSE)

BOOL serial::Settings(SCHNL *port,int *baud,PARITY *parity,char *bits,char *stop)

SERIAL PORT USER CONFIGURATION (DIALOG)

Description: Sets up the user GUI based configuration routine (a dialog box).

Passed parameters: appl - current application handle

Returned value: none

```
void serial::Dialog(HANDLE appl)
```

The serial class is implemented in SERIAL.CPP. Since this class is only a thinly veiled wrapper to standard non-overlapped Windows API functions, the implementation is straightforward WIN32 programming (see the article "Serial Communications in WIN32" in the MSDN library etc.). The only exception to this is the routine WaitChar. The current implementation uses a polling approach that allows other threads to operate. In pseudo-code the routine is:

```
BOOL WaitChar(DWORD timeout)
```

```
WHILE (not timed out) AND (character not received)
    get Windows port input queue status
    IF (no input) THEN
        give up time slice
    ENDIF
ENDWHILE
IF (timed out) THEN
    return is FALSE
ELSE
    return is TRUE
ENDIF
```

3.2.8.3 TCP/IP Socket Link

This link provides the same communication functional interface via a socket. It is defined in TCPIPLINK.H and implemented in TCPIPLINK.CPP. The primary design decisions made were:

1. The link uses a TCP/IP based stream using Windows sockets version 1.1 (Note: this is the version supported on all Win 32 platforms including CE).
2. Both server and client use a single connection to do their business and then disconnect. There are no free running multi-connection server type components.
3. Since the socket stream does not delineate messages, the link level interface must.
4. It can use RAS or DUN resources to make connections when

configured to do so (see section 4).

The last of these requires that the message stream be encoded in a manner similar to that used in the serial communication link:

```
end of packet:      <ESC><0x02>
end of message:    <ESC><0x20>
escape:           <ESC><ESC>      [not a control symbol]
```

A "Send" then simply takes the application's message (or sub-message), encodes it and sends it on its way with a socket send. The implementation of receive is a bit more complicated and as in the serial implementation uses two parts, a free running receive thread and the Receive function itself. The scheme is summarized in the following Figure.

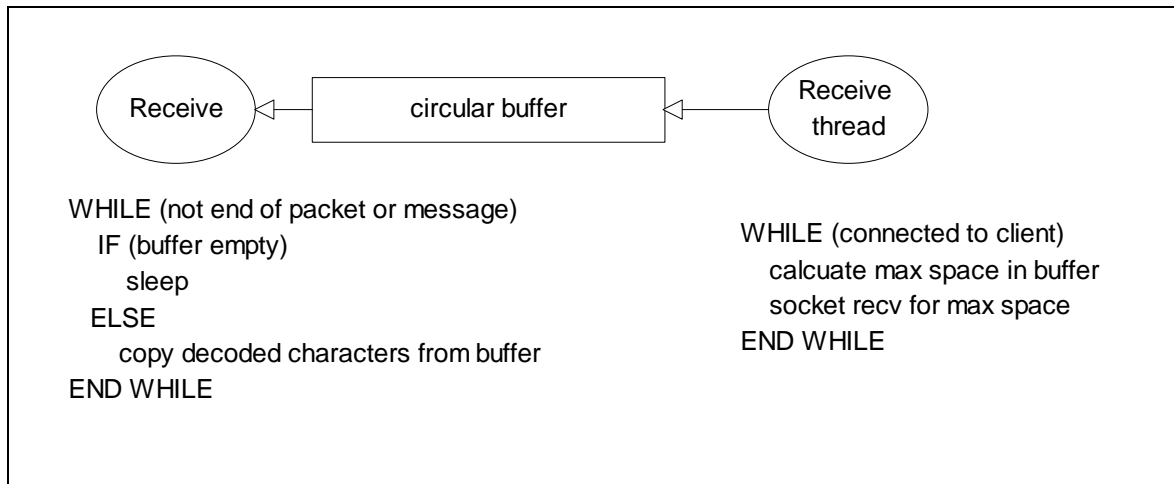


Figure 9, Receive implementation

3.2.9 File Manager

The file manager framework class provides a complete set of file operations for a control application. Characteristics are:

1. Files are either text, binary or raw (i.e. no internal known format)
2. Line oriented I/O provided for text files (line ends with carriage return - line feed).
3. A blob orientation is used for binary or raw files.
4. Text and binary file access only operates on the "data section" of file.

5. Raw file access operates on the file's entire content (i.e. it assumes no knowledge of file structure).
6. A manager takes care of only one file at a time

Because of the last of these characteristics, multiple file managers may be active at the same time. Often file managers are dynamically created and destroyed (new and delete).

To be thread safe, the file manager opens files using shared access. If write or read - write access is requested then the file is only opened if no one else, i.e. another thread has it open. After opening access to the file is denied to another other file manager, no matter what access they are requesting. If read only access is requested the file is opened only if no one else has access or the other access(es) are read only. After opening, access to the file is denied to another file manager requesting write or read - write access.

The class is defined in FILEMAN.H. A detailed definition of the public function interface follows:

OPEN AN EXISTING OR NEW FILE

Description: Provides a means to open a file. If a file name of *.xxx or *.* is specified then a common dialog box is used to get the filename. Otherwise, the specified file is opened. If the file exists then it is placed into the appropriate mode, text or binary. Key characteristics of open are

1. When the action is read, the file type must match the type being opened.
2. A write or write & read action always causes the file to be created.

Passed parameters: filename - pointer to file name string
action - file operation to perform (read, write, write & read, foreign edit)
type - file type for write (TEXTF, BINARYF or RAWF, default is TEXTF)
title - pointer to title string (default is NULL)
open_name - pointer to string buffer to store the file name if file name was *.xxx or *.* (default is NULL)

Returned value: Open was successful (TRUE) or not (FALSE)

BOOL file_manager::Open(char *filename, file_opn action, char

*title,char * open_name)

CLOSE THE CURRENTLY OPEN FILE

Description: Closes the currently open file if any.

Passed parameter: none

Returned value: operation successful (TRUE) or not (FALSE)

BOOL file_manager::Close(void)

WRITE TO THE CURRENTLY OPEN FILE

Description: Appends a blob of data to the currently open binary or raw file

Passed parameters: blob - pointer to data to write
no - number of bytes to write

Returned value: write successful (TRUE) or not (FALSE)

BOOL file_mangager::Write(unsigned char *blob,short int no)

WRITE A LINE TO THE CURRENTLY OPEN FILE

Description: Appends a line to the currently open text file.

Passed parameters: line = pointer to line to write
no - number of characters in the line

Returned value: write successful (TRUE) or not (FALSE)

BOOL file_manager::WriteLn(TCHAR *line,short int no)

READ A LINE FROM THE CURRENTLY OPEN FILE

Description: Reads a data blob from the open binary or raw file.

Passed parameter: buf - pointer to callers input buffer
max - maximum number of bytes that can be accepted

Returned value: number of bytes read

short int file_manager::Read(unsigned char *buf,int max)

READ A LINE FROM THE CURRENTLY OPEN FILE

Description: Reads the current line in the open text file.

Passed parameter: buf - pointer to callers input buffer
 max - maximum number of characters that can
 be accepted

Returned value: number of characters read

short int file_manager::ReadLn(TCHAR *buf,int max)

DELETE THE CURRENTLY OPEN FILE

Description: Deletes the currently open file if any.

Passed parameter: none

Returned value: operation successful (TRUE) or not (FALSE)

BOOL file_manager::Delete(void)

REWIND THE CURRENTLY OPEN FILE

Description: Places the currently open file, if any, back to the first line.

Passed parameter: none

Returned value: operation successful (TRUE) or not (FALSE)

BOOL file_manager::Rewind(void)

MOVE TO THE END OF THE CURRENTLY OPEN FILE

Description: Places the currently open file, if any, at the end of the file.

Passed parameter: none

Returned value: operation successful (TRUE) or not (FALSE)

BOOL file_manager::GoToEnd(void)

CHECK FOR END OF FILE

Description: Checks for the end of file condition.

Passed parameters: none

Returned value: end of file (TRUE) or not (FALSE)

BOOL file_manager::EndFile(void)

RENAME A FILE

Description: Changes the currently open files name to the name indicated. If the new name includes a path then this path will be used. Otherwise, the path of the current (to be replaced) file name will be used.

NOTE: The file will be positioned at its beginning upon return from this call.

Passed parameter: new_name - pointer to desired new file name

Returned value: operation succeeded (TRUE) or not (FALSE)

BOOL file_manager::Rename(char * new_name)

The file manager class is implemented in FILEMAN.CPP. It is primarily just a wrapper for combining the common file dialogs and Win32 file operations. An example use of the file manager, the test class used to initially check its operation, can be found in FILETEST.CPP.

3.2.10 Console

The console framework class provides the means to display text in a child window's client area. It is actually implemented as two classes. The base class provides the guts of console output. It supports a logical screen in RAM (screen) that can be vertically and horizontally scrolled to map to the physical display. It operates as a registered message object within the framework. This allows it to handle any user/Window's required changes such as position scrolling, content copying etc.

Specific "display rules" are implemented in console type classes that inherit this base class. They provide the user display operator (<<) that includes the display rules for that type. They use the protected functions in the base class to actually output to the client area.

The base class is defined in CONSOLE.H and the derived class currently implemented in TTYCON.H. A more extensive definition of publicly available functions follows:

Base Class Functions:

INITIALIZE OBJECT

Description: Initializes the console object. This has the following effects:

1. A child window is created and positioned in the based on the pos parameter (desktop only, always top - center on CE platform)
2. The system cursor is changed to WAIT (desktop only)
3. The input to the child window (mouse or keyboard) is disabled

Called parameters: title - title for console window
pos - position indicator

Returned values: window handle if successful or NULL

HWND console::Init(TCHAR *title,unsigned char pos)

REFRESH THE SCREEN

Description: Refreshes the entire client area screen.

Called parameters: repaint - called for WM_PAINT or not

Returned values: none

void console::Refresh(BOOL repaint)

SCREEN WIDTH

Description: Provides the width of the current screen.

Called parameter: none

Returned value: none

int console::Width(void)

SCROLL LEFT/RIGHT

Description: Provides means to scroll left or right.

Called parameters: scroll - columns right (positive) or left (negative)

Returned value: none

```
void console::Hscroll(int scroll)
```

SCROLL UP/DOWN

Description: Provides the means to scroll up and down.

Passed parameters: scroll - rows up (positive) or down
(negative) to scroll

Returned value: none

```
void console::Vscroll(int scroll)
```

COPY SCREEN CONTENT TO CLIPBOARD

Description: Copies the entire content of the RAM screen to the clipboard.

Passed parameters: none

Returned value: none

```
void console::Copy(void)
```

GET CONSOLE TYPE

Description: Returns the console type.

Passed parameter: none

Returned value: console type

```
con_type console::Type(void)
```

WAIT FOR USER TO CLOSE CONSOLE'S CHILD WINDOW

Description: Sets the console's child window to accept input, changes the cursor and waits for the user to close the console's child window.

Passed parameters: none

Returned value: none

```
void console::WaitUserClose(void)
```

CLOSE THE CONSOLE (WINDOW AND ALL)

Description: Closes the console including its child window.

Passed parameters: none

Returned value: none

void console::Close(void)

Derived Class Functions:

<< (char) OPERATOR OVERLOAD

Description: Provides an overloaded "shift out" operator for a single character for the raw console class.

Passed parameter: chr - the unsigned char to output

Returned value: reference to this object

console& ttycon::operator<< (char chr)

<< (*char) OPERATOR OVERLOAD

Description: Provides an overloaded "shift out" operator for a string for the console class.

Passed parameter: cpt - pointer to character string

Returned value: reference to this object

console& ttycon::operator<< (char *cpt)

This is a limited implementation since it only provides for logical screen (maintained in the two dimension array "screen"} to physical screen mapping. It does not provide for logical screen to virtual screen (maintained in a file) mapping through vertical scrolling. The variables vtop, top, cur and left_edge are indexes into the screen array that provide the means of mapping the RAM screen to the actual display and for keeping track of the currently filled region (see Figure 5). The core public functions are just manipulations of these indexes:

Hscroll: move left_edge

Vscroll: move top

```

Refresh:  clear the screen
          for (each line from top to min of max_lns or cur)
              output the line
          endfor

```

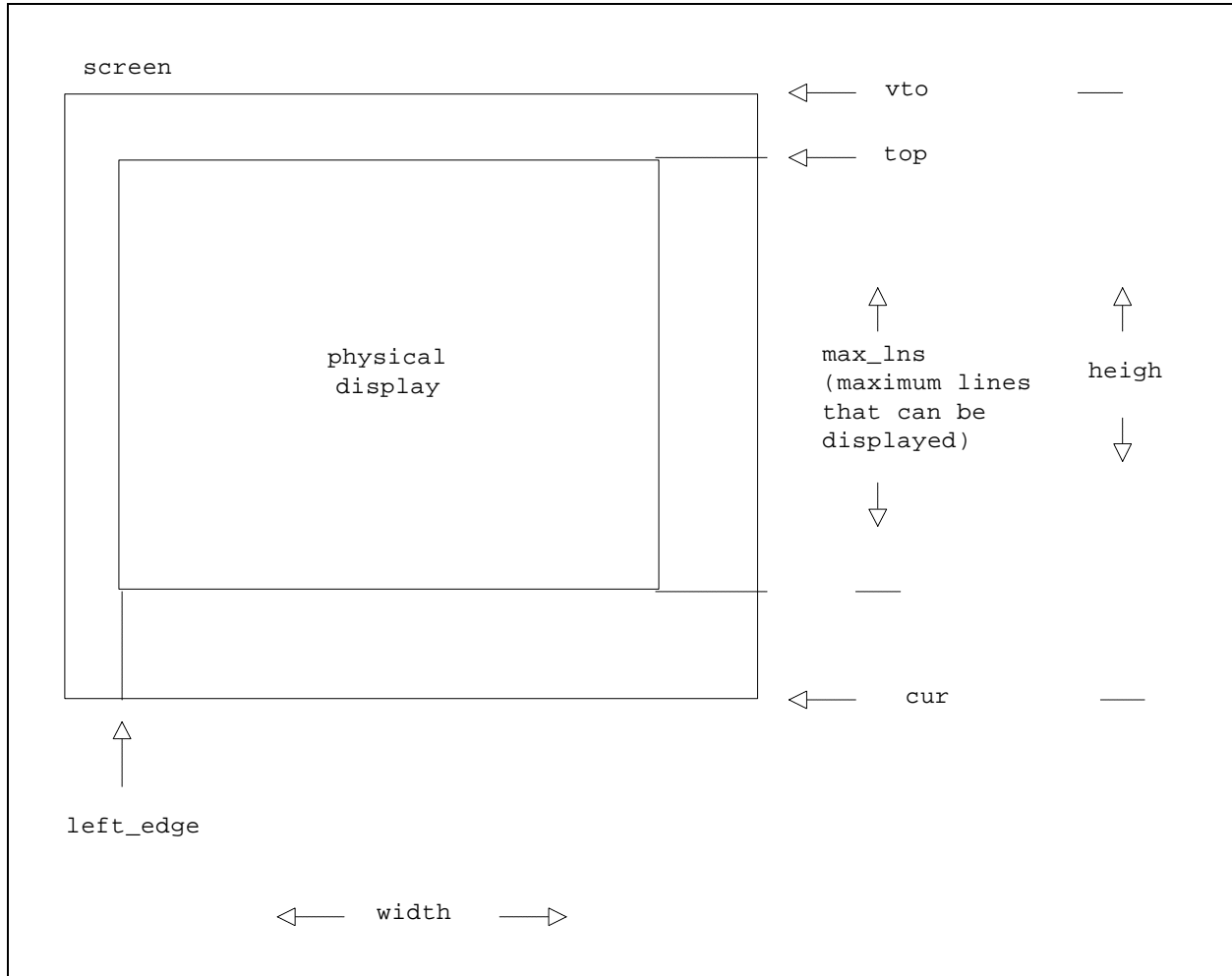


Figure 10, Display Layout

Adding a character to the screen uses `cur` to set the line (or row) and the `screen[cur][0]` to get the character position (or col) within the line. A line wrap or line feed requires:

```

increment cur by 1 line
IF (cur equals vtop) THEN
    increment vtop by 1 line
ENDIF
IF (distance from top to cur is greater than max_lns) THEN

```

```

        increment top by 1 line
        Refresh
ELSE
    display line
ENDIF

```

The copy routine also uses these variables to determine the size of the filled region in screen and to copy it to the clipboard. In pseudo-code this is:

```

OpenClipboard
EmptyClipboard
set size to 0
FOR (vtop to cur)
    add the number of characters on line (screen[row][0])to size
ENDFOR
allocate memory for size bytes
FOR (vtop to cur)
    copy the line to allocated memory
ENDFOR
SetClipboard
CloseClipboard

```

The current implementation creates a child window that uses approximately half (desktop) to 2/3rd (CE palm-sized) of the application's client area. It automatically wraps lines that are wider than it's window's width. Since the child window is created within the console's "main" thread, it contains its own message loop and window message procedure. Because of the nature of this display the window has some very distinctive characteristics:

1. A child window is created and positioned based on the position parameter provided (desktop only, position 0 is the only one supported under CE)
 - a. position 0 - top, center
 - b. position 1 - top, left
 - c. position 2 - top, right
 - d. position 3 - bottom, left
 - e. position 4 - bottom, right
2. The system cursor is changed to WAIT (desktop only)
3. The input to the child window (mouse or keyboard) is disabled

When the component using the console window is done (i.e. it has displayed all the data it has) then the routine WaitUserClose provides the means to wait for a user to close the child window. This routine:

1. Sets the console's child window to accept input.
2. Changes the cursor (desktop only)
3. Waits for the user to close the console's child window in a poll loop.

4 CONFIGURATION FILE

The configuration file, `GENERIC.CFG`, includes definitions for key pieces of information that the framework uses. Supported definitions are:

`COMM ERROR LOG=TRUE`

This specifies that the communication links should keep error logs. The default case is that error logging is disabled.

`port="name",type,parameter list`

This specifies a communication port to be activated by the program. If the specified port can not be opened then an error message box is displayed. If no port definition exists then an error message box with the message "Could not open communications link" appears. The parameter list is depended on the type of port, `SERIAL` or `TCPIP`. For a `SERIAL` port the parameters are:

baud rate - any level Windows serial rate (bps). If the baud rate is not specified then the default baud rate is used.

remote - whether the connection is with a remote device (`TRUE`) or not (`FALSE`), default is `FALSE`

"modem name" - The modem name is optional and is only used if remote is `TRUE`. If remote is `TRUE` and no name is provided the first modem found will be used.

For a `TCPIP` port the parameters are:

IP address - the IP address of the socket to connect with if a client or the IP address of this node if a server

TCP port - the TCP port address of the socket to connect with if a client or the TCP port address of this node if a server

type port - specified if this port should operate as a client or a server

For `CLIENT` type `TCPIP` ports, additional configuration information can be provided in the following statements:

`port name connection="DUN connection name"`

This specifies the name of the dial up network connection to use. The default is no DUN connection is used (i.e. have a "hard wired" connection such as a LAN or WAN).

```
port name user="user name"
```

This specifies the user name to use with the DUN connection. The default is a blank name (which causes DUN to ask the user).

```
port name password="password"
```

This specifies the password to use with the DUN connection. The default is a blank password (which causes DUN to ask the user).

An example file follows:

```
Filename: GENERIC.CFG
```

```
Date: 3/23/99
```

```
Start Data
```

```
COMM ERROR LOG=TRUE
```

```
port="COM1",SERIAL,57600,FALSE
```

```
port="Device as server",TCPIP,192.0.0.1,2000,CLIENT
```

```
Device as server connection="My Connection"
```

```
Device as server user="me"
```

```
Device as server password="my password"
```

```
End data
```

APPENDIX A, WINDOW'S CE CONSIDERATIONS

Window's CE supports a sub-set of the complete Win 32 APIs found in the desktop environment. In addition, physical differences such as display size, I/O devices supported etc, require some implementation restrictions or conditional compile statements. These are summarized below:

1. Basic restrictions:
 - a. Any use of semaphores was replaced with events.
 - b. TAPI usage was restricted to version 1.5.
 - c. All string handling uses TCHAR based definitions and functions.
 - d. Winsock usage was restricted to version 1.1 functions.
2. Most of the frameworks source files have a few conditional compile statements for CE and/or CE palm-sized:
 - a. Generically a CE environment is detected using "UNDER_CE". This definition is automatically provided in Visual C++ when working with CE.
 - b. The palm-sized only CE environment is detected using "PALM-SIZE". This definition must be added to the pre-processor symbols.
3. The file manager was re-written to use Win 32 file functions since the C run-time file functions are not supported under CE (NOTE: With the VC++ 6.0 toolkit the C run-time file functions are supported).
4. The TCP/IP socket link required some conditional compile statements for CE because of an undocumented quirk in the Microsoft socket function inet_addr.

Because the directory/file access on CE devices (especially palm-sized) has a number of restrictions, the installation scheme is completely regimented. The scheme is:

```
executable(s): \My Programs\Generic
support file(s): \My Documents\Generic
help files: \Windows
```

Building operational components which will port across environments requires:

1. Use string functions/definitions that support either ASCII or unicode format (TCHAR etc.)
2. Only use C run time library functions that are documented as

being supported for CE.

3. Only use Win 32 API functions that are documented as being CE 2.0 supported.
4. Be careful with the location of dynamically selected non-byte variable boundaries. The MIPS or SH3 processor require different considerations than the Intel processors. This is especially a potential problem when parsing a byte stream received from a device (see LNKSTAT.CPP for an example of handling this kind of problem).

We have found that it is useful to build the first prototype of a development cycle for the desktop environment, even if the only target will be a CE device. If done carefully, this allows the development team to implement and test in a "full feature" environment and then quickly port and tune on the CE device.

APPENDIX B, KNOWN ISSUES

The known issues with the base control application portion (i.e. non-Cimetrix) of the C++ Framework/toolkit are:

1. Communication performance will degrade with certain system configurations. First generation CE hand-held devices with slow processors, e.g. the 33 MHz 16 bit MIPS in the Philips Velo 1, provide very low levels of through put no matter what baud rate. Newer devices, even palm-sized devices, are significantly better. Modem latency can cause decreases in performance, especially with intense bi-directional communication traffic that is composed of short messages, i.e. the control application workload. In this case, TCPIP is not a high performance protocol for short bi-directional communication traffic (i.e. point to point serial will do substantially better even with significantly less link bandwidth).
2. The components GUIs and the on-line help provided with the generic control application are crude. They are intended to serve as place fillers, thought they are correct and fully functional.
3. The handling of unnamed remote serial ports could be significantly improved by checking for a port match when looking for modems rather than just taking the first one available.
4. Windows 2000 testing to date has been limited to compatibility checks of the generic control application and a relatively short, ten hours, test run with the verification control application test build. Both applications were produced under NT 4.0. In addition, the development wizards, both project and component, have been run through basic install and operation checks under Windows 2000. There have been no incompatibilities found with Windows 2000 Professional.
5. Limited testing has been performed using the point to point serial protocol and a USB port with a Serial - USB adapter. This has indicated no problems but the amount of testing at this point is inadequate to verify this configuration.
6. The project file created by the Project Wizard is in Visual C++ 5.0 format. With one exception, this can be converted to 6.0 format by Visual Studio with no loss of information. The one exception is the Windows CE configuration, hand-held or

palm-sized. As a result a CE targeted project may need to be manually set to the correct configuration initially after conversion.