Important User Information

Solid state equipment has operational characteristics differing from those of electromechanical equipment. "Safety Guidelines for the Application, Installation, and Maintenance of Solid State Controls" (Publication SGI-1.1) describes some important differences between solid state equipment and hard-wired electromechanical devices. Because of this difference, and because of the wide variety of uses for solid state equipment, all persons responsible for applying this equipment must satisfy themselves that each intended application of this equipment is acceptable.

In no event will Rockwell Automation be responsible or liable for indirect or consequential damages resulting from the use or application of this equipment.

The examples and diagrams in this manual are included solely for illustrative purposes. Because of the many variables and requirements associated with any particular installation, Rockwell Automation cannot assume responsibility or liability for actual use based on the examples and diagrams.

No patent liability is assumed by Rockwell Automation with respect to use of the information, circuits, equipment, or software described in this manual.

Reproduction of the contents of this manual, in whole or in part, without written permission of Rockwell Automation is prohibited.

Throughout this manual, we use notes to make you aware of safety considerations.

**ATTENTION:** Identifies information about practices or circumstances that can lead to personal injury or death, property damage, or economic loss.

**Important:** Identifies information that is especially important for successful application and understanding of the product.
# Table of Contents

## Using this Manual
- Who Should Use This Manual ................................................... P-1
- Purpose of this Manual .............................................................. P-1
- Contents of this Manual ........................................................... P-1
- Manual Conventions .................................................................. P-2
- Allen-Bradley Support ............................................................... P-2

## Introduction to the RAC6182
### Chapter 1
- Hardware Architecture .............................................................. 1-1
- Software Architecture ............................................................. 1-6

## Developing CE Drivers and Applications for the RAC6182
### Chapter 2
- General Considerations ........................................................... 2-1
- Setting Up the Development System ......................................... 2-3

## RAC6182 CE SDK
### Chapter 3
- Overview .................................................................................... 3-1
- Files in the C/C++ Development Kit ......................................... 3-2

## RAC6182-Specific Extensions to the CE API
### Chapter 4
- Functions for Digital Output Control ......................................... 4-1
- Functions to Read from and Write to the Bezel EEPROM .......... 4-7
- Function for Watchdog Timer Control ........................................ 4-11
- Functions for Use in Custom Keypad Handlers ......................... 4-12
- Streams Interface for Keypad Driver Control ......................... 4-14
- Streams Interface for Touchscreen Control.............................. 4-14
- Functions for LED Control .......................................................... 4-14
- Functions for Use in PCI Device Drivers .................................. 4-18
- Sample Code for a Simple PCI Slot Device .............................. 4-25
- Functions for OS Update ............................................................ 4-27
- Function for Registry Flush ....................................................... 4-27
- Function to Adjust Allocation of DRAM ..................................... 4-29
- Functions to Get/Set Misc Parameters ..................................... 4-30
- Functions for Accessing System Timers .................................... 4-33
- Functions for Accessing the Hardware Monitor ....................... 4-40
- Functions for Accessing Retentive Memory ............................. 4-50
- Streams Interface for Serial Ports ............................................. 4-55
- Application Interface to Output Debug Messages .................... 4-57
Appendix A

Operating System Files .............................................................A-1
Memory Usage ..........................................................................A-2
Connecting an External Debug Monitor.................................A-3

Index
Using this Manual

Read this preface to familiarize yourself with the rest of the manual. The preface covers the following topics:

- who should use this manual
- the purpose of the manual
- contents of the manual
- conventions used in this manual
- Allen-Bradley support

Who Should Use This Manual

Use this manual if you are responsible for developing application software to run on the 6182 Windows CE Industrial Computer.

Purpose of this Manual

This manual is a user guide for the Software Development Kit for the 6182 Windows CE Industrial Computer. It gives an overview of the system and provides detailed information about the contents of the software development kit.

Contents of this Manual

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>Describes the purpose, background, and scope of this manual. Also specifies the intended audience.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Introduction to the RAC6182</td>
<td>Provides an overview of the 6182 Computer and describes the hardware and operating system software.</td>
</tr>
<tr>
<td>2</td>
<td>Developing CE Drivers and Applications for the RAC6182</td>
<td>Provides general guidelines for programmers. Provides detailed procedures for setting up the development system and installing the RAC6182 SDK.</td>
</tr>
<tr>
<td>3</td>
<td>RAC6182 CE SDK</td>
<td>Provides an overview of the 6182 CE SDK.</td>
</tr>
<tr>
<td>4</td>
<td>RAC6182-Specific Extensions to the CE API</td>
<td>Provides detailed descriptions of the 6182 functions.</td>
</tr>
<tr>
<td>Appendix A</td>
<td>Provides file lists, memory usage information, instructions for using an external debug monitor.</td>
<td></td>
</tr>
</tbody>
</table>
Manual Conventions

The following conventions are used throughout this manual:

- Bulleted lists such as this one provide information, not procedural steps.
- Numbered lists provide sequential steps or hierarchical information.

Allen-Bradley Support

Allen-Bradley offers support services worldwide, with over 75 Sales/Support Offices, 512 authorized Distributors and 260 authorized Systems Integrators located throughout the United States alone, plus Allen-Bradley representatives in every major country in the world.

Local Product Support

Contact your local Allen-Bradley representative for:

- sales and order support
- product technical training
- warranty support
- support service agreements

Technical Product Assistance

If you need to contact Allen-Bradley for technical assistance, please review the information in the System Troubleshooting chapter first. Then call your local Allen-Bradley representative or contact Allen-Bradley technical support at (440) 646-5800.

For additional product information and a description of the technical services available, visit the Rockwell Automation/Allen-Bradley Internet site at http://www.ab.com.
Introduction to the RAC6182

CPU

The system processor is a QED RM5231 embodying the popular MIPS 4300 RISC architecture. The RM5231 has a 225MHz clock speed, 64-bit internal registers and a built in floating point unit.

The system processor communicates directly with various memory devices, a V320 bus controller, and a super I/O controller by way of a 32-bit system address and data bus.

The V320 bus controller manages system dynamic RAM for the 5231. It also provides a bridge from the system bus to the PCI bus. In addition, it provides system timers.
System Timers
At least one (and in some cases, depending on the hardware revision level, more than one) programmable hardware timer is available at the application layer.

Memory Devices

Disk-On-Chip Flash ROM
The Disk-On-Chip device (commonly called the “DOC”) is a flash ROM (32MB to 256MB, field upgradable) that emulates a disk device. The Disk-On-Chip device has two partitions or logical storage areas. One partition provides non-volatile storage for the Windows CE operating system image. The other partition supports a FAT16 (DOS compatible) file system, in which application programs and data can be stored.

Boot ROM
The boot code resides in a 256KB boot ROM. The boot code includes a “Boot Loader” that loads the Windows CE image from the Disk-On-Chip into DRAM at system startup.

The boot ROM is segmented into two 128K blocks, each of which contains boot code. A jumper on the CPU board (J2) selects between the 2 blocks. The lower block (selected when the jumper is across pins 1 and 2) contains the primary boot code. The upper block (selected when the jumper is across pins 2 and 3) contains boot code with debug support. The upper block boot code performs extended power on self testing (POST), disables restoration of user persistent registry items, and enables debug output on COM2.

DRAM
The RAC6182 uses industry standard 3.3V, PC100 and SPD compliant, non-ECC dynamic RAM, packaged in a DIMM. The DIMM may be 32MB to 256 MB, and is field upgradable. The DRAM provides a fast access volatile storage space for data and program code.

The Operating System uses part of the RAM for a RAMDISK and the other part for normal system memory. The RAMDISK portion is known as the Object Store and provides specialized storage for the Windows CE Registry and Windows CE system databases. The Windows CE Control Panel System Properties tool has a slider control that allows a user to determine how the RAM is allocated between RAMDISK Storage and system memory. The slider control is factory set for a 50/50 split. Application programs can control RAM allocation with the Windows CE system call SetSystemMemoryDivision (see Microsoft’s documentation of the CE API for details).
Retentive Memory (Battery Backed RAM)

A 128 KB non-volatile memory provides application accessible storage for state information and data logging operations. A Lithium battery with a 10-year shelf life provides for long term data retention. It is recommended that applications using the retentive memory monitor battery voltage; this is easily done with system calls to the RAC6182’s hardware monitor. If the battery voltage should start to decline, the battery should be replaced. Battery replacement is facilitated by the presence of a “super-cap”, which provides sufficient current to sustain the memory data for up to eight hours following a power down of the system.

Super I/O

The National PC97317 super I/O chip provides several key functions.

PS/2 Keyboard and Mouse Ports

The super I/O chip provides two PS/2 ports to support connection of an external keyboard and/or mouse.

Serial Ports

The super I/O chip provides two 16550A compatible serial ports. The CE operating system identifies these ports as COM1 and COM2. Applications can control both COM ports via standard WIN32 API function calls.

COM1 is provided with a DB9 connector and supports RS232 (TXD, RXD, RTS, CTS) and RS422/485 (TX422+, TX422-, RX422+, RX422-) signaling at standard communication data rates up to 115 Kbps. The configuration (RS-232/RS-422/485) for this port is software-selectable. COM1 serves as the primary serial port for application use; it remains available to applications for serial I/O even when debug mode is enabled.

COM2 is provided with a DB9 connector and supports “standard” RS232 signaling only. COM2 may serve either as an application RS232 port or as a debugging port. When the operating system is in debugging mode it will output debug messages to COM2 port. An application developer or device tester can utilize these messages to determine the current state of the operating system, or to identify problems such as device failures or application exceptions.

Parallel Port

The super I/O chip provides one bi-directional parallel port to support connection of a local printer or other peripheral device.
Real Time Clock

The super I/O chip provides a Real Time Clock which keeps the system date and time.

Watch Dog

The super I/O chip provides a watchdog timer that can be used to trigger a system reset.

At system initialization, the watchdog is disabled. It can be enabled by an application. Once the watchdog is enabled, one or more applications must periodically “tag” (restart) it to prevent it from timing out. If the watchdog times out, a system reset (warm-boot) is initiated. Once the system has been restarted, an application can inquire about the event that caused the restart and learn that the watchdog timed out. A condition code indicating this remains latched and detectable by software until it is cleared by a cold-boot.

The watchdog timer has a maximum resolution of 1 msec.

GP I/O Subsystem

The super I/O chip provides a bridge between the system bus and a general purpose I/O bus. This general purpose I/O bus supports several additional devices.

Relay Output

A relay output is provided for application level control of an external device.

Hardware Monitor

An application accessible hardware monitor provides real-time temperature, voltage and battery monitoring. Thresholds for warnings can be established by application programs. Applications also have access to the system LEDs.

Keypad

Certain configurations of the RAC6182 provide function keys, a numeric keypad and cursor control keys integrated into the front bezel. The number of function keys can vary. Some function keys are re-legendable.

Extended software support for the bezel keypad is provided with the RAC6182 Windows CE operating system in the form of a keypad handler DLL. The keypad handler intercepts and operates on codes produced by the keypad driver before passing them to the application with current focus. The keypad handler can optionally re-map keys (assign different virtual key codes) and effect specialized processing such as the generation of key macros (strings of virtual key codes) or the launching of a new program from single key strokes. The standard
keypad handler can be replaced by a custom keypad handler to provide special key code mappings.

**Touch Screen**

An integral, resistive analog touch screen with a serial controller provides mouse-like operator input. The touch screen is a factory installed option associated with an integral display.

**PCI Subsystem**

The V320 chip provides a bridge between the 32-bit system address/data bus and the PCI bus. Several devices are attached to the PCI bus.

**Display Controller**

An MQ200 video controller, configured as device z on PCI bus 0, supports bezel mounted LCDs as well as external monitors. The MQ200 provides 2 MB of video RAM.

**Table A**

<table>
<thead>
<tr>
<th>Display Options</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Integral 8&quot; Diagonal Display</strong></td>
</tr>
<tr>
<td>Type</td>
</tr>
<tr>
<td>STN LCD with LVDS digital interface</td>
</tr>
<tr>
<td>TFT LCD with LVDS digital interface</td>
</tr>
<tr>
<td>STN LCD with LVDS digital interface</td>
</tr>
</tbody>
</table>

*The bezel mounted LCD display is a factory installed option.

**USB Ports**

A USB0672 USB Controller Chip, configured as device x on PCI bus 0, provides basic OHCI host support for up to two USB peripheral devices. This basic support will facilitate use of various USB keyboards, printers, bar code readers, etc. when the appropriate device specific drivers are available.

**On-board Ethernet**

An Intel 82559 Fast Ethernet Multifunction Controller, configured as device x on PCI bus 0, provides Ethernet communications support via one 10/100-BaseT RJ45 port.

**PCMCIA Slots**

A PCMCIA slot connector supports 2 Type II PC Cards or 1 Type III PC Card. The PC Cards can be memory or I/O devices. The system supports concurrent operation of PC Cards as follows: One for memory
expansion and one for communications; one or two for memory expansion; one or two for communications.

**PCI Slot**

One half-length PCI slot provides an expansion capability for communication and I/O. The PCI slot can accommodate a large assortment of specialized and commercially available PCI add-in cards when suitable drivers are available. The device installed in the PCI slot will be device x on PCI bus 1.

**Software Architecture**

**RAC6182 Windows CE OS**

The initial release of the RAC6182 provided Windows CE Version 2.12 with the latest service packs. Currently, the RAC6182 is provided with Windows CE V3.0

The system software includes the following components:

- Hardware Initialization and Boot Loader, situated in the boot ROM
- Windows CE Kernel with adaptations (Hardware Adaptation Layer customized for the RAC6182 hardware, Built-in ISRs), situated in the boot image stored in the operating system partition of the Disk-On-Chip
- Windows CE Default Registry, which is part of the boot image. (A persistent registry, containing information relative to specific configurations, is maintained in the file system and merged with the default registry at boot.)
- Windows CE Modules and Device Drivers (File system support, ...), implemented as part of the boot image or as files (dlls, exes, etc.) stored in the FAT16 partition of the Disk-On-Chip
- GUI Desktop Shell, implemented
- Control Panel and System Configuration/View Tools

**Boot Sequence**

The boot code in the Boot ROM gets control of the microprocessor at power-on, initializes the hardware, performs power-on self-tests (POST), and moves the compressed Windows CE operating system image from the boot partition of the Disk-On-Chip persistent storage device into DRAM. Several seconds are required for the decompression and copy operation. Finally, the boot loader jumps to the start address of the Windows CE image and control passes to the Windows CE operating system. Windows CE then loads drivers, including the driver for the Disk-On-Chip FAT16 file system (on the "storage card" partition), restores the registry, establishes the video modes, and finally loads the start-up applications into memory and runs them.
The operating system image that is loaded resides in the operating system partition of the Disk-On-Chip device. However, before loading the Disk-On-Chip boot image, the boot code checks for the presence of a PCMCIA memory device that is capable of supplying a boot image. The flow diagram that follows describes the boot sequence.

**Boot Sequence Overview Rev D**

**Hardware Initialization**

The boot code first initializes the CPU and V320 system controller; it then tests video RAM, which it needs to use for stack and heap until the system DRAM is fully available. If debug is enabled, the debug serial port is initialized and a 1 second delay is introduced to give the baud rate
generator time to stabilize. Finally, system DRAM is tested. In the interest of reducing boot time, this test is limited to an address check; no attempt is made to identify bit errors at given addresses.

Tests for Boot Devices

When hardware testing has been completed, the boot code starts looking for PCMCIA devices capable of supplying a compressed operating system image.

The boot code first checks for the presence of a PCMCIA ATA memory device. If it finds such a device with a FAT16 file system containing a file recognizable by its name as a compressed boot image, it will attempt to use that image. The image will be tested for validity. If valid it will be used to overwrite any existing image on the Disk-On-Chip. Operating system loading will then commence.

Note: The boot process configures the PCI controller and any NE2000 Ethernet device in the PCI slot to the extent that, after boot, applications will be able to obtain necessary address and interrupt information by querying the device. This basic POST setup does not necessarily configure all configuration space registers such a device may use. Though many PCI devices will not need any other configuration space registers configured after boot, some have special power-management or other registers to configure. These registers vary widely by card and should be setup by application after boot using Win API calls to access PCI configuration space.

Load of Compressed Operating System

The boot code reads the compressed operating system image from the Disk-On-Chip operating system partition, decompresses it and loads it into memory. (It loads the executable operating system code into program memory and a default system registry into the RAMDISK section of memory.) Control then passes to the operating system image in memory.

“Cold Boot”

The operating system begins a “cold boot” by loading the driver for the FAT file system on the Disk-On-Chip.

It then attempts to find the primary persistent registry file. If this file is not present, it attempts to find the backup persistent registry file. If no persistent registry file is found, system boot continues with the default registry already in memory.

If a persistent registry file is found, the system merges the default operating system registry and this saved persistent registry, saved persistent registry items taking precedence.
“Warm Boot”

After the registry merge, a “warm boot” is begun. Control passes to the operating system kernel, which can now use the registry image to initialize various subsystems. The file system drivers, the graphical subsystem drivers, serial, network, PS/2, USB, and other device drivers are loaded and initialized.

The Windows CE Registry

The Windows CE Registry contains application and system configuration data. The Control Panel provides the user interfaces for managing the system settings that are configurable by the user. Applications access the Registry via the Win32 API. Application developers can manipulate the registry using the tools in Microsoft’s Widows CE Toolkit for Visual C++ 6.0, Windows CE Toolkit for Visual Basic 6.0 or Windows Embedded Tools V3.0.

The default Registry resides in the operating system image in the Disk-On-Chip. During runtime, the Registry is loaded into and resides in RAM in the Object Store (RAMDISK).

When the system is powered-on, the registry is restored from Flash Memory to DRAM during “cold boot”.

At system shutdown, a persistent copy of the registry is written to flash memory by a FlushRegistry() operation. Likewise, when the built-in Control Panel application, used to manage system settings, is closed, a persistent copy of the registry is written using a FlushRegistry() call. FlushRegistry() may also be called by an application.

Table B

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\storage\Registry.rlz</td>
<td>Primary persistent Registry file</td>
</tr>
<tr>
<td>\storage\regbak.rlz</td>
<td>Backup persistent Registry file</td>
</tr>
<tr>
<td>\storage\regtemp.rlz</td>
<td>Temporary persistent Registry file</td>
</tr>
</tbody>
</table>

These files are accessed by RegistryFlush and other operating system functions. Applications should never access them directly. The primary and backup persistent Registry files have the read-only, hidden, and system attribute bits set to prevent accidental corruption or deletion by application.

The only time these files should be deliberately touched is during a condition of suspected Registry corruption, wherein, the user decides to revert to the default Registry. Deleting both files and restarting will revert to the default Registry.
The operating system boot process is responsible for merging the default operating system Registry keys with the keys from the persistent Registry. If the same keys exist, preference is given to the persistent registry file. A few default keys are exceptions to this rule and are bypassed during the merge; e.g. the O/S version number is acquired from the O/S image.

The process of merging default and persistent registry information allows operating system upgrades to add new registry keys and values and have these be used in addition to any saved registry state. Since the saved registry information has precedence, users’ saved registry keys for control panel applets and other operating system items will be maintained even in the case of operating system upgrades.

On the other hand, the priority given to persistent registry information over default operating system registry information makes it possible for applications or users to cause problems with operating system startup by changing the wrong registry keys. When manipulating the RAC6182’s CE Registry applications and users should exercise the same degree of caution that would required in the case of a Windows 9x or NT device.

**Important:** Since some applications and drivers only read the Registry at start-up, some registry changes made by applications will have no effect until the RAC6182 is re-started.

**Policies for When Registry Flushing Occurs**

Control panel applets supplied with the operating system have been customized to automatically flush the registry upon exiting the applet. This allows users to change typical control panel settings such as network, device name, screen saver, etc. and have these be flushed without having to manually issue a registry flush to save these. Since the flush occurs on applet exit as an optimization, users just need to remember to close the applet after making changes for the automatic flush to occur. Due to the inner workings of the applets, it is not feasible to only flush on applet close if a value was changed, so a flush occurs on applet close even if no registry values were actually altered.

Other applications such as Internet Explorer, remote networking, and any third-party packaged applications are not customizable in this fashion and hence changes they make to the registry will not be persistent until some other application flushes the registry. To address this, two features of the operating system are present.

First, an executable regflush.exe supplied with the system may be manually executed by a user at any time to flush the registry to persistent storage; this application simply calls RegistryFlush(). Second, upon a controlled shutdown requested by an application through the power/shutdown driver results in an automatic flush of the registry after applications have signaled that their cleanup is complete and before the hardware is actually shutdown or reset.
During an uncontrolled shutdown (i.e. hard-power down), the system does not have enough time to flush the registry to persistent storage. Therefore, the registry must have been flushed by one of the means described above or else changes to the registry since the last flush will be lost. It is recommended that the controlled shutdown procedure be used for shutdown even if other registry flushing by applications is in place.

Local File Systems

The RAC6182 Windows CE operating system provides support for two separate local file systems. A DOS compatible FAT16 file system is implemented in one of the two Disk-On-Chip partitions; accordingly, its files are persistent. A RAM file system (RAMDISK or Object Store) is implemented in that part of the system DRAM reserved for it. The files in the RAM file system are not persistent.

The FAT16 and RAM file systems can be viewed and manipulated by the Windows Explorer utility. Within the Windows Explorer, these systems appear as parts of one larger system. That is, they appear as directories under “My Computer”. The FAT16 file system appears as “Storage Card”, while the RAM file system includes several directories, including the most important, the “\Windows” directory, where system binaries are stored.
### Table C
**RAM File System**

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\Temp</td>
<td>Not used</td>
</tr>
<tr>
<td>\My Documents</td>
<td>Not used</td>
</tr>
<tr>
<td>\Program Files</td>
<td>Contains links (shortcuts) to certain system executables</td>
</tr>
<tr>
<td>\Program Files\Communications</td>
<td>Contains links (shortcuts) to certain system executables</td>
</tr>
<tr>
<td>\Windows</td>
<td>Contains system executables (<em>.exe), dynamic link libraries (</em>.dll), fonts (*.ttf), etc. making up the Windows CE operating system</td>
</tr>
<tr>
<td>\Windows\Programs</td>
<td>Contains links (shortcuts) to certain executables in \Windows</td>
</tr>
<tr>
<td>\Windows\Programs\Communication</td>
<td>Contains links (shortcuts) to certain executables in \Windows</td>
</tr>
<tr>
<td>\Windows\Desktop</td>
<td>Contains links (shortcuts) that define the contents of the Windows Desktop</td>
</tr>
<tr>
<td>\Windows\Favorites</td>
<td>Not used</td>
</tr>
<tr>
<td>\Windows\Fonts</td>
<td>Not used</td>
</tr>
<tr>
<td>\Windows\Recent</td>
<td>Not used</td>
</tr>
<tr>
<td>\Windows\Startup</td>
<td>Not used</td>
</tr>
</tbody>
</table>

The FAT16 (persistent) file system, “\Storage Card”, is organized as follows:

### Table D
**FAT16 File System**

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\Storage Card</td>
<td>Contains backups of the system registry and the system exceptions log. Applications should be stored here or in subdirectories created here.</td>
</tr>
<tr>
<td>\Storage Card\Temp</td>
<td></td>
</tr>
<tr>
<td>\Storage Card\Windows\Desktop</td>
<td>Contains links to certain system executables</td>
</tr>
<tr>
<td>\Storage Card\Windows\Programs</td>
<td>Contains links to certain system executables</td>
</tr>
</tbody>
</table>
Input Device Handlers

Touch Screen

The RAC6182’s display can be equipped with a high resolution resistive touch screen. The Windows CE operating system incorporates a driver for the touch screen.

A user interface is provided to enable touch screen configuration and calibration. Touch screen calibration values are stored in the registry.

Keyboards

The RAC6182 is designed to take key input from multiple sources. Support is present in the operating system for a standard PS/2 keyboard, a standard USB keyboard, and a bezel keypad. The key input drivers are designed to permit any one of these devices to function alone and to permit a bezel keypad to function together with a PS/2 or USB keyboard.

Note: There is no support for both a PS/2 and a USB keyboard simultaneously connected

The PS/2 and USB keyboards can be individually enabled or disabled using control panel applets.

The Windows CE architecture dictates that one keyboard device and one only may be loaded by the GWES.EXE subsystem, and that this device will be responsible for the default mappings of scan codes to virtual keys, virtual keys to virtual keys, and virtual keys to Unicode characters.

Other keyboard devices are supported as device drivers loaded by DEVICE.EXE. These drivers submit virtual keys to the primary keyboard driver and use its mapping capability. Multiple key input devices share modifier (SHIFT, CONTROL, ALT) states.

The primary (GWES) driver in the RAC6182 is the driver that handles the PS/2 keyboard and mouse ports. The USB keyboard and keypad drivers are dependent upon this driver for high level key input processing. The following table identifies the various drivers that constitute the keyboard input subsystem.

Table E
Drivers for the Keyboard Input Subsystem

<table>
<thead>
<tr>
<th>Driver</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\windows\kbdmouse.dll</td>
<td>PS/2 keyboard and mouse driver, loaded by GWES.EXE at startup. Responsible for low level PS/2 related items and scan code to virtual key mappings for the PS/2 keyboard. Responsible for default virtual key to virtual key mappings based on modifier keys and for virtual key to Unicode mappings, for all key input devices.</td>
</tr>
</tbody>
</table>
**Driver** | **Description**
---|---
`\windows\usbhid.dll` | USB Human Interface Device driver, loaded by DEVICE.EXE upon insertion of a USB Human Interface Device. Handles USB keyboard and mouse. Responsible for low-level USB related items and scan code to virtual key mappings for USB keyboard. Submits virtual key codes to kbdmouse.dll.

`\windows\keypad.dll` | RAC6182 specific keypad driver, loaded by DEVICE.EXE at startup. Handles low-level keypad input and scan code to virtual key mapping. Submits virtual keys to Rockwell supplied keypad handler for mapping and submits virtual keys to kbdmouse.dll for virtual key to Unicode mappings.

`\windows\khstub.dll` | Keypad handler stub. This DLL is loaded by keypad.dll if no Rockwell supplied keypad handler is present. The stub returns a default scan code to virtual key mapping table for one current model of keypad and defers all virtual key mapping to the kbdmouse.dll driver.

`\storage\card\kh.dll` | Rockwell supplied keypad handler, loaded by keypad.dll on boot. Responsible for mapping virtual keys from the keypad into other virtual keys, macros, or other actions. Any virtual keys returned by the keypad handler’s mappings will still use kbdmouse.dll for mapping virtual keys into Unicode. The name of this file may be overridden with an alternate keypad handler name via a registry key. If key [HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Keypad] contains a REG_SZ value named “KeypadHandlerName”, its value will instead be used when loading the keypad handler.
The following figure schematizes the functional relations between the various drivers in the keyboard subsystem.

As can be seen from the table and the accompanying schematic, the functions of the RAC6182 bezel keypad are supported by two separate software components: a keypad driver, and a keypad handler.

**Keypad Driver**

The keypad driver supports low level functions associated with standard keyboards (e.g., generation of auto-repeat sequences, mappings of scan codes to Windows virtual key codes, etc.) and a number of Rockwell proprietary features:

- Support for multiple types of keypads. Different keypads may have different scan code to virtual key mappings.
- Support for non-standard keys, for example, the programmable function keys K1 – K16 and the ‘+’ key (unshifted press results in a ‘+’, shifted press results in a ‘*’).
- Support for mapping single key presses into multiple key macros at the virtual key level.
Support for assignment of special functions to key operations by application programs.

Support for a ‘single-key’ mode, in which keystrokes are processed one at a time. Following an initial key-down event, any other key-down or key-events will be ignored until the key-up event corresponding to the initial key-down event has been detected and processed.

Support for a ‘hold-off’ mode, in which successive strokes of a given key occurring within a given time period will be ignored.

After the keypad driver is loaded by device.exe at system start-up, it attempts to read the keypad ID from the bezel EEPROM. If it does not find a valid ID, it exits. Otherwise, using the keypad ID, the driver locates an entry in the CE system registry that points to the current scan code to virtual code translation table for the keypad.

The keypad driver then attempts to load the keypad handler and verify that it supports a set of callback functions that the driver requires it to have. If the keypad handler dynamic link library is not present or does not contain all the necessary callback functions, a default keypad handler stub is loaded. This handler stub implements all the necessary callbacks and information for mapping one particular model of keypad, but it cannot handle changing key mappings, macros, actions, or other models of keypad.

When a key on the keypad is pressed or released, the keypad processor sends two codes to the keypad driver. One code is a scan code corresponding to the key pressed or released; the other is an event code identifying the type of event (key up or key down). Using the current translation table, the driver converts the scan code into a Windows Virtual key code. The driver maintains the modifier, auto-repeat, and multiple-keys states.

The driver does additional processing of key events to determine if these events meet the conditions of repeat mode, hold-off mode or single-key mode, provided these modes are enabled.

Once it has finished its low level processing, the driver calls the keypad handler function KhTranslateVkey(), passing the virtual key code to this function. The keypad handler returns an array of translated virtual code(s).

Finally, the driver calls a Win32 API function kbd_event() to pass the key events to the GWES keyboard driver.

Keypad Handler

The Rockwell supplied keypad handler is an optional software component that can be replaced with a stub or with another keypad handler designed for a specific application. The handler operates on Windows Virtual Key codes supplied by the keypad driver. It can
perform translations of Virtual Key codes before the keypad driver passes these codes to the main keyboard driver for final processing. Thus, it functions as an intermediate processor between the keypad driver and the main keyboard driver.

The keypad handler maintains its own key mapping and attribute tables separate from those maintained by the keypad driver. It can maintain these tables, in the system registry, system file storage, or wherever else the implementer of the keypad handler chooses. Although these mapping and attribute tables will be used by the driver, they are placed under the control of the handler to facilitate changes in mapping or attribute information and to facilitate the support of various keypads. With this scheme, new features and functions can be accommodated without modifications to the driver or other operating system level modules.

The handler also maintains global configuration data for the keypad, including auto-repeat settings, single key and hold off mode settings, etc.

The keypad handler is loaded and initialized by the keypad driver, and the handler must be able to respond to an initial query from the driver for its key mapping and attribute information.

Once the handler has been initialized by the driver, it is ready to accept additional calls from the driver to map any incoming virtual key down presses or releases that are currently valid (subject to the constraints of hold off and single key mode, which are enforced by the driver). The keypad handler may perform some action based on the key code passed (for example, it may launch an application), it may expand a key code into a sequence of codes (implementing a macro definition), it may filter the code, re-map it, etc. Alternatively, it may defer mapping of the virtual key to the normal keyboard driver.

In addition to being called back for key presses, the keypad handler will be called back when the global configuration settings for the keypad driver are changed. The keypad handler or some other application may change the settings of the keypad driver using the streams interface to be discussed later. When this occurs, the keypad handler is called back to ensure that it is aware of the changes.

**Registry keys used by KHSTUB.EXE**

The operating system includes a simple keypad handler stub which may be used when the more sophisticated capabilities in the Rockwell handler are not required. This stub defers all mapping from the virtual key level up to the main keyboard driver. The registry keys kghstub uses to obtain keypad mapping and other information are documented here in case application developers wish to use the same keys.
Global key setting information is listed here by key and value.

[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Keypad\Params\TypematicRepeat]
  "Enabled"    REG_DWORD which is 1 for enabled, 0 for disabled
  "RepeatDelay" REG_DWORD of initial repeat delay in ms.
  "RepeatRate" REG_DWORD of subsequent repeat delay in ms.

[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Keypad\Params\SingleKeyMode]
  "Enabled"    REG_DWORD which is 1 for enabled, 0 for disabled
  "AbortEnabled" REG_DWORD which is 1 for enabled, 0 for disabled

[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Keypad\Params\HoldoffMode]
  "Enabled"    REG_DWORD which is 1 for enabled, 0 for disabled
  "HoldoffTime" REG_DWORD of time in ms. for key hold-off

Keypad ID specific items include scan code to virtual key mappings and the attribute flags. The key name contains the keypad ID printed as a %04X value to reference the correct keypad mappings. The value names contain the scan code number printed as a %02X value. A sample for a keypad with ID 0x0A5C is given here.

[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Keypad\Keypads\0A5C]
  "VirtualKey00" REG_DWORD giving the virtual key code for scan code 0x00
  "Attributes00" REG_DWORD giving the key attributes for scan code 0x00
  "VirtualKey6A" REG_DWORD giving the virtual key code for scan code 0x6A
  "Attributes6A" REG_DWORD giving the key attributes for scan code 0x6A

**Bezel ID EEPROM**

The bezel EEPROM provides a total of 256 bytes of non-volatile storage. The first 128 bytes are reserved for use by the CE operating system. The remaining 128 bytes are available for use by application programs.

The bezel EEPROM is used by the operating system to identify components of the bezel and traits specific to that bezel. These components and traits may vary from unit to unit and so are appropriately kept with the bezel. This allows interchange of bezels without reprogramming or recalibration. The bezel configuration information stored in this EEPROM is used by three major CE subsystems: the video driver, the touch screen driver, and the keypad driver.

**Video Data**

The video driver uses a 16 bit value in the bezel EEPROM to identify the LCD by model. This ID is used to reference an area of registry for settings used for any LCD panel of that model, such as resolution, interface type, and panel timings.

The video driver also stores minimum and maximum values for contrast and brightness in the bezel EEPROM. These values define limit values presented on the control panel. They are stored in the EEPROM rather than registry since individual panels of the same model may have variances that require individual adjustments based on experimental data.
The current contrast and brightness values are individual user preferences rather than traits of the panel, so are stored in registry rather than in the bezel EEPROM. However, if the registry contrast or brightness value is not present or not in the min/max range specified for the panel by the bezel EEPROM data, a default value is used from the bezel EEPROM.

**Touch screen data**

The touch screen driver uses a 16 bit value in the bezel EEPROM to identify the model of touch screen. This is used to determine what type of touch screen interface and decoding logic to use. Also, the bezel EEPROM is used to store touch calibration data, since calibration data will be specific to particular touch screens even of the same model.

**Keypad data**

The keypad driver uses a 16 bit value in the bezel EEPROM to identify the model of the keypad. This ID is used to determine what keypad scan code to virtual key mapping information is obtained from the keypad handler for the driver to use in decoding keys.

**Layout of the Bezel EEPROM**

The following table shows the layout of the bezel EEPROM. Assignments within the lower 128 bytes (system area) are subject to change. However, the upper 128 bytes (beginning at offset 0x80) are guaranteed to be available for use by applications needing non-volatile storage.
### Table F
**Layout of the Bezel EEPROM**

<table>
<thead>
<tr>
<th>Addresses (Hex)</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00-0x01</td>
<td>16 bit CRC of the remainder of bezel EEPROM data</td>
</tr>
<tr>
<td>0x02-0x0F</td>
<td>Reserved for future system use</td>
</tr>
<tr>
<td>0x10-0x11</td>
<td>Touch screen ID (0=4 wire res, 1=5 wire res, 0xffff = none)</td>
</tr>
<tr>
<td>0x12-0x13</td>
<td>Magic cookie to tell if calibration data is valid</td>
</tr>
<tr>
<td>0x14-0x1F</td>
<td>Reserved for future touch screen driver use</td>
</tr>
<tr>
<td>0x20-0x2F</td>
<td>Touch screen calibration information</td>
</tr>
<tr>
<td>0x30-0x31</td>
<td>Keypad ID (0 = 56 key pad, 1 = 84 key pad, 0xffff = no keypad)</td>
</tr>
<tr>
<td>0x32-0x3F</td>
<td>Reserved for future keypad driver use</td>
</tr>
<tr>
<td>0x40-0x41</td>
<td>LCD panel ID (0 = 7&quot; STN, 1 = 12&quot; TFT, 0xffff = no LCD)</td>
</tr>
<tr>
<td>0x42-0x43</td>
<td>Minimum allowable contrast for this particular panel</td>
</tr>
<tr>
<td>0x44</td>
<td>Maximum allowable contrast for this particular panel</td>
</tr>
<tr>
<td>0x45</td>
<td>Default contrast for this particular panel if registry contrast is invalid</td>
</tr>
<tr>
<td>0x46</td>
<td>Minimum allowable brightness for this particular panel</td>
</tr>
<tr>
<td>0x47</td>
<td>Maximum allowable brightness for this particular panel</td>
</tr>
<tr>
<td>0x48-0x5F</td>
<td>Default brightness for this particular panel if registry brightness is invalid</td>
</tr>
<tr>
<td>0x60-0x7F</td>
<td>Reserved for future operating system use</td>
</tr>
<tr>
<td>0x80-0xFF</td>
<td>Application area. Will not be used by operating system</td>
</tr>
</tbody>
</table>

### PCI Bus

PCI bus 0 contains the onboard Ethernet, video, USB, and PCMCIA controllers.

PCI bus 1 contains the PCI slot. From a PCI configuration standpoint, the virtual slot number of a device plugged in the slot is 1.

The operating system supports basic configuration, interrupt control, memory management and IO access for PCI cards plugged into this slot. The operating system does not support bus-mastering by the PCI slot device.

### PCMCIA

New or upgraded components of application programs and the operating system can be copied from the PCMCIA memory card to Disk-On-Chip flash memory to replace and upgrade the existing components.
In the Windows Explorer, the PCMCIA Memory Card will show up as an icon named “Storage Card2”.

**Application Run Time Environment**

**Path**

The notion of a path to executable files is much the same as with any other Windows or DOS system. However, unlike other systems, which refer to an environment variable for path settings, Windows CE utilizes a registry entry. Thus, the path can be set only by editing the value of the registry key \HKLM\Loader\SystemPath. Note the use of spaces to separate items in the path list, as in the following example:

“\storage card\bin\ \storage card\ \storage card2\bin\ \storage card2\ . . .”

**Launching Applications At Start-Up**

The Windows CE Registry entries at key HKLM\init determine the programs that are started during system initialization, and the order in which they are started. The Windows CE Platform Builder development tool (not part of the Windows CE Toolkits for Visual C++ 6.0 and Visual Basic 6.0) is used to establish these Registry entries.

**Table G**

**RAC6182 Launch Order**

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Program or File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Launch10</td>
<td>shell.exe</td>
<td>Start the shell</td>
</tr>
<tr>
<td>Launch20</td>
<td>device.exe</td>
<td>Load and start the device drivers</td>
</tr>
<tr>
<td>Launch30</td>
<td>postdevice.exe</td>
<td>Start post device driver processing . . .</td>
</tr>
<tr>
<td>Depend30</td>
<td>14 00</td>
<td>when device.exe signals complete</td>
</tr>
<tr>
<td>Launch40</td>
<td>gwes.exe</td>
<td>Start graphics and events subsystems . . .</td>
</tr>
<tr>
<td>Depend40</td>
<td>1e 00</td>
<td>when postdevice.exe signals complete</td>
</tr>
<tr>
<td>Launch50</td>
<td>explorer.exe</td>
<td>Start Windows Explorer . . .</td>
</tr>
<tr>
<td>Depend50</td>
<td>14 00 28 00</td>
<td>when device.exe and gwes.exe complete startup</td>
</tr>
<tr>
<td>Launch90</td>
<td>(an OEM executable)</td>
<td>Start OEM executable . . .</td>
</tr>
<tr>
<td>Depend90</td>
<td>1e 00 28 00</td>
<td>when postdevice.exe and gwes.exe complete startup</td>
</tr>
</tbody>
</table>

Launch90 provides a launch point at startup for an OEM that assures that the device drivers, TCP/IP, registry and GUI functions are up and running.

Explorer is launched during initialization because it handles the GUI shell, taskbar, running items in \windows\startup, etc. Unlike other executable files, Windows Explorer does not properly signal that it has
completed startup, so dependencies should not be placed directly on explorer.exe. Consequently, the start menu, taskbar, etc. may still be drawing when oemstartup.exe is called.

Although there is a \windows\startup folder in the file system, the placement of a shortcut in this folder in order to start the associated application automatically at system startup is not recommended. The folder \windows\startup is RAM based, and its contents will not persist from one operating session to the next.

The solution is to place shortcuts in \Storage Card\Windows or in a directory under it. In a normal system initialization sequence, everything in \Storage Card\Windows\ (in the persistent file system), including subdirectories and their contents, is copied to \Windows (in the RAM filesystem) following the startup of gwes.exe. This copy operation is not performed only when the system has been placed in diagnostic mode, either by the installation of a jumper on the system board (see information about the boot ROM elsewhere in this manual), or by an application, using a call to the system function rm_SetParameter (see the description of this function elsewhere in this manual).

**Process Priorities**

All executable files start in user mode. Any application can change to kernel mode or back with the Windows CE SetKMode() call. The only known exception is nk.exe, which is started first and doesn't follow the same rules.

**System Shutdown**

The system supports a soft reset and provides a shut-down indicator in non-volatile memory.
Developing CE Drivers and Applications for the RAC6182

General Considerations

There are two general considerations for developing drivers and applications for the RAC6182:

- Distributing and installing applications
- Persistence considerations

Application Distribution and Installation

Application programs for the RAC6182 will consist of EXE and DLL files that will reside in the FAT partition of the Disk-On-Chip. They will be installed much like applications for Windows desktop operating systems.

Typically, a RAC6182 CE application will be distributed as a package containing the run-time components, in compressed form, and an executable “installation script” that manages the installation process. When the installation script (typically “Setup.exe”) is run, the run-time components will be decompressed and moved to their assigned folder(s), desktop icons and start menu entries will be created, and the system registry will be edited to register the application’s components and associated parameters. Finally, an uninstall script will be created and saved.

Microsoft’s InstallShield tool is recommended for packaging applications for distribution. This tool alleviates some of the difficulties associated with the development of installation scripts and imposes a familiar “look and feel” on the installation process.

The application developer should give some thought to the means to be used for distributing the installation script. Generally, there are two means available: CDROM and the internet.
Installing the Application

Once the user has obtained an installation script by one of these methods and the script resides on the user’s local desktop PC, he or she may use any of three methods to install the application on the RAC6182.

Perform a remote installation by running the script on a PC host that is connected to the RAC6182 using Data Exchange.

Copy the script from a PC host using Data Exchange or from a PCMCIA ATA memory card to the “storage card” folder on the RAC6182 and run the script on the RAC6182.

Run the script directly from a PCMCIA ATA memory card on the RAC6182.

Remote Installations

The install package can be quite large and the decompression process can consume high levels of memory, so remote installation is an attractive option. Data Exchange, using CeAppMgr.exe on the host PC and WCEload.exe on the RAC6182, supports remote installation.

Application Upgrades

The application developer should make appropriate provisions for issuing application upgrades from the beginning, adopting good practice for source version control, bug reporting, etc. When upgrades are required, typically by the desire to add new features or to implement bug fixes, decisions will have to be made relating to the notification of users and the distribution of the upgrades. Considerations for the distribution and installation of application upgrades are exactly the same as those discussed above for initial distribution and installation.

Persistence Considerations

Installation of a new application program on the RAC6182 typically adds a new icon to the Windows CE Desktop and sometimes a new entry in the Start Menu, in order to enable the user to launch the new program or to launch it automatically. Shortcuts in the folder “\Windows\Desktop” create the icons on the desktop. Shortcuts and subfolders in the folder “\Windows\Programs” form the Start Menu. A shortcut in the folder “\Windows\Startup” will automatically launch a program at startup. A control panel applet that was added by an application has a file extension *.CPL and resides in the folder “\Windows.”

All this appears very Windows-like and ordinary until one considers that the “\Windows” folder is effectively a RAM disk that is recreated when cold-started; i.e. it is not persistent. When the operating system boots it creates a new file system including “\Windows” and that effectively removes all traces of the end-user applications that once existed. With that in mind, special considerations are necessary for applications on the
RAC6182 and all similar embedded devices since the Icons, the Start Menu, and application-provided Control Panel Applets must be re-created at startup.

The solution is to place shortcuts in \Storage Card\Windows or in a directory under it. In a normal system initialization sequence, everything in \Storage Card\Windows\ (in the persistent filesystem), including subdirectories and their contents, is copied to \Windows (in the RAM filesystem) following the startup of gwes.exe. (For further information see “Launching Applications At Start-Up” above.)

Typically, development will take place on an x86 machine running a Microsoft Win32 operating system and Microsoft cross development tools. The development system will be connected to the target RAC6182 by Ethernet or serial link, and MIPS binary files generated on the development system will be downloaded to the target for testing and debugging.

While for the most part the Microsoft development tools will run on Windows 95, Windows 98 and Windows 2000, certain special functions, like emulation of the target platform on the x86 host, are available to the developer only with Windows NT 4.0.

Application development can be carried out using either C/C++ or Basic. Note that the C/C++ development system normally produces MIPS binaries that are directly executable on the RAC6182, while the Basic development system produces application code modules (.vb files) that must be run with the help of a Basic interpreter on the RAC6182. The Basic development system includes an application installer that bundles the application code module with the interpreter (consisting of MIPS executables) so that all the components necessary for program execution will be properly installed on the RAC6182. Device driver developers should plan to use C/C++.

**Setting Up the Host Machine for C/C++ Development**

First, Microsoft Windows CE Services (Active Sync) must be installed on the host system. This package provides utilities needed to download applications to the RAC6182 and to support a number of remote development tools. Windows CE Services is provided on CDROM with the RAC6182. The RAC6182 User’s Manual (Chapter 14) contains detailed information about installation.

Next, the following Microsoft development tools must be installed on the host system, in the order given:

- Microsoft Visual C++ 6.0 (from Visual Studio 6.0, Professional or Enterprise Edition)
Windows CE Toolkit for Visual C++ 6.0
Platform SDK for H/PC – MIPSFP (from Windows CE Toolkit for Visual C++ 6.0)

Or,

Embedded Visual C++ 6.0 (from Microsoft Embedded Visual Tools 3.0)
Platform SDK for H/PC – MIPSFP (from Microsoft Embedded Visual Tools 3.0)

Note: The user of the Windows CE Toolkit for Visual C++ 6.0 should note that a special configuration step is necessary to work around a known limitation of that package. (See configuration details below.)

While the Windows CE Toolkit for Visual C++ 6.0 is an extension of the Visual C++ 6.0 tool from Visual Studio and depends upon it, Embedded Visual C++ 6.0 is a stand-alone tool that does not require the installation of Visual C++ 6.0 from Visual Studio. However, Embedded Visual C++ 6.0 supports development for CE platforms only, and not for Windows desktop operating systems. Developers contemplating ports to CE of applications written originally for Windows desktop operating systems will probably want the support for both CE and desktop OS development that is available with Visual C++ 6.0 extended with the Windows CE Toolkit.

On the other hand, Microsoft Embedded Visual Tools 3.0 is available without charge, except for a nominal shipping and handling charge. Accordingly, it is an economical tool for developers of new CE only applications.

Finally, while Embedded Visual Tools 3.0 is not integrated with Visual Studio’s tools, it can co-exist with these tools.

Device driver developers should consider also installing the Microsoft Windows CE Platform Builder, which contains support for kernel level CE development that is not found in the other toolkits. However, Platform builder is not necessary for most driver development work.

Details of the installation procedures are beyond the scope of this manual. Please follow the instructions provided by Microsoft.

Finally, the RAC6182 SDK should be installed. (See detailed instructions below.)
Setting Up the Host Machine for Basic Development

First, Microsoft Windows CE Services (Active Sync) must be installed on the host system. This package provides utilities needed to download applications to the RAC6182, and to support a number of remote development tools. Windows CE Services is provided on CDROM with the RAC6182. The RAC6182 User’s Manual (Chapter 14) contains detailed information about installation.
Next, the following Microsoft tools must be installed on the development platform in the order given:

Microsoft Visual Basic 6.0 (from Visual Studio 6.0, Professional or Enterprise Edition)

Windows CE Toolkit for Visual Basic 6.0

Platform SDK for H/PC - MIPSFP (from Windows CE Toolkit for Visual Basic 6.0)

Or,

Embedded Visual Basic (from Microsoft Embedded Visual Tools 3.0)

Platform SDK for H/PC - MIPSFP (from Microsoft Embedded Visual Tools 3.0)

Note: While the Windows CE Toolkit for Visual Basic 6.0 is an extension of the Visual Basic 6.0 tool from Visual Studio and depends upon it, Embedded Visual Basic is a stand-alone tool that does not require the installation of Visual Basic 6.0 from Visual Studio. However, Embedded Visual Basic supports development for CE platforms only, and not for Windows desktop operating systems. Developers contemplating ports to CE of applications written originally for Windows desktop operating systems will probably want the support for both CE and desktop OS development that is available with Visual Basic 6.0 extended with the Windows CE Toolkit.

On the other hand, Microsoft Embedded Visual Tools 3.0 is available without charge, except for a nominal shipping and handling charge. Accordingly, it is an economical tool for developers of new CE only applications.

Finally, while Embedded Visual Tools 3.0 is not integrated with Visual Studio’s tools, it can co-exist with these tools.

Details of the installation procedures are beyond the scope of this manual. Please follow the instructions provided by Microsoft.

Finally, install the RAC6182 SDK.

Installing the RAC6182 SDK

Installing the RAC6182 SDK is the final step in setting up the development system.
The RAC6182 SDK is provided on CDROM. The CDROM contains two different development kits, one for the RAC6182 with CE V2.12, and one for the RAC6182 with CE V3.0. Either of these development kits may be installed separately. Most users will want to install the SDK for CE V3.0.

Before installing the SDK for CE V3.0, it is recommended that any existing installation of the SDK for CE V2.12 be removed. All RAC6182s with CE V2.12 can be easily upgraded to CE V3.0. Applications developed to run on V2.12 should port to V3.0 without difficulty, and all subsequent development can proceed on V3.0.

Each development kit includes two executable files, one containing an SDK for Visual C++ and the other containing an SDK for Visual Basic. Both can be installed on the same machine if desired; however, it is not necessary to install both.

The following installation instructions pertain to the SDK for CE V3.0. The procedure for installing the CE V2.12 SDK is similar.

To install, insert the RAC6182 CDROM in the host machine’s CDROM drive (normally drive D:) and from the Windows GUI issue the following instructions:

Start Run D:\Win CE 3.0\RAC6182VB-2.00.exe (for Visual Basic installation)

or

Start Run D:\Win CE 3.0\RAC6182VC-2.00.exe (for Visual C++ installation)

The installer will be prompted for acceptance of a license agreement. Following that, the SDK should install itself automatically on the host machine without further operator intervention. When installation is complete the following message should appear: “The SDK was successfully installed”. The installer must press the “Done” button in the install window to exit.
The RAC6182 SDK CDROM contains the following additional files:

- VBSDKReadme.txt - Information about the Visual Basic SDK
- VCSDKReadme.txt - Information about the Visual C++ SDK
- 6182api.txt - Visual Basic function definitions file

**Configuration**

After installing the SDK for CE V2.12, some special configuration is required. This applies only to the SDK for CE V2.12. The following steps are not required for the CE V3.0 SDK.

Users of Embedded Visual C++ will want to set up some directories immediately after installing the RAC6182 SDK. In the Embedded Visual C++ IDE

Under Tools Options Directories Include Files

Add \c:\Windows CE Tools\WCE212\RAC6182\User Files\Vc\Inc

Under Tools Options Directories Library Files

Add \c:\Windows CE Tools\WCE212\RAC6182\User Files\Vc\Lib\mipsfp

Users of Windows CE Toolkit for Visual C++ 6.0 will want to set up the following directories immediately after installing the RAC6182 SDK. In the Visual C++ IDE

Under Tools Options Directories Executable Files

Change \c:\Windows CE Tools\WCE212\bin

TO \c:\Windows CE Tools\WCE211\bin

Under Tools Options Directories Include Files

Add \c:\Windows CE Tools\WCE212\RAC6182\User Files\Vc\Inc

Under Tools Options Directories Library Files

Add \c:\Windows CE Tools\WCE212\RAC6182\User Files\Vc\Lib\mipsfp
No special configuration is necessary for Embedded Visual Basic.
Overview

The RAC6182 SDK provides developers with access to an extensive set of functions that are specific to the RAC6182 hardware and constitute extensions of the standard Windows CE API. These functions, like the standard Windows CE functions, are implemented in the C language and can be called directly from C or C++ programs.

Basic programs can also call these functions. However, Basic programs must declare the functions in the proper form before invoking them. For example, a Basic program might contain the following:

```
' Basic declaration of the C library function Watchdog_Tag()
Const WATCHDOG_OK = &0
Declare Function Watchdog_Tag Lib "watchdog.lib" Alias "Watchdog_Tag"
(ByVal dwTimeout As Long) As Long

... 
'
Invocation of the function
if (Watchdog_Tag(3000) equ WATCHDOG_OK) then
  ' do something
endif
```

A file called “6182api.txt” is included in the RAC6182 SDK. This file includes Basic declarations for all the constants, data structures and functions associated with the RAC6182 SDK C language libraries. Basic programmers can copy declarations from this file into their programs as needed, just as they can copy the declarations for the standard CE functions from a Microsoft provided file called “winceapi.txt”.

C/C++ language developers should note that the headers included in the RAC6182 SDK contain conditionals that allow them to be included in C and C++ modules without modification. A C++ program should include a #define __cplusplus directive prior to an #include <sdk_header> directive, or else the __cplusplus macro should be defined on the compiler command line. Users of the Microsoft Visual C++ 6.0 IDE will not have to make any special provisions in their programs, since this IDE makes C++ is default for a new project and defines this macro for them.
On the other hand, users of this IDE who wish to write in standard C should keep in mind that this default situation will require all standard C modules to be conditionally bracketed in the same way that the headers in the SDK are bracketed. For example:

```c
#ifdef __cplusplus
extern "C" {
#endif
/* C code goes here */
#ifdef __cplusplus
}
#endif
```

### Files in the C/C++ Development Kit

<table>
<thead>
<tr>
<th>Component</th>
<th>C Header</th>
<th>Static Library</th>
<th>Dynamic Library (Part of the OS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aux Microcontroller</td>
<td>atmelapi.h</td>
<td>ATMEL.lib</td>
<td>atmel.dll</td>
</tr>
<tr>
<td>Bezel EEPROM</td>
<td>bezeeleeprom.h</td>
<td>BEZEL.lib</td>
<td>bezel.dll</td>
</tr>
<tr>
<td>PCI Subsystem</td>
<td>Ceddk.h</td>
<td>ceddk.lib</td>
<td>ceddk.dll</td>
</tr>
<tr>
<td>CE Shell</td>
<td>shlobj.h</td>
<td>ceshell.lib</td>
<td>ceshell.dll</td>
</tr>
<tr>
<td>Digital Outputs</td>
<td>DiagnosticOutputAPI.h</td>
<td>DiagnosticOutput.lib</td>
<td>DiagnosticOutput.dll</td>
</tr>
<tr>
<td>Voltage and Temperature Monitor</td>
<td>HardwareMonitorAPI.h</td>
<td>HardwareMonitor.lib</td>
<td>HardwareMonitor.dll</td>
</tr>
<tr>
<td>Keypad Driver</td>
<td>KeypadAPI.h</td>
<td></td>
<td>keypad.dll</td>
</tr>
<tr>
<td>Keypad Handler</td>
<td>khapi.h</td>
<td></td>
<td>kh.dll , khstub.dll</td>
</tr>
<tr>
<td>Misc. System</td>
<td>miscsystem.h</td>
<td>MISCSYSTEM.lib</td>
<td>miscsystem.dll</td>
</tr>
<tr>
<td>LEDs</td>
<td>nledapi.h</td>
<td>(none required)</td>
<td>coredll.dll</td>
</tr>
<tr>
<td>Operating System</td>
<td>OSUpdateAPI.h</td>
<td>OSUPdate.lib</td>
<td>OSUpdate.dll</td>
</tr>
<tr>
<td>Serial Ports</td>
<td>othersdk.h</td>
<td></td>
<td>coredll.dll</td>
</tr>
<tr>
<td>Keypad Mapping</td>
<td>RAC6182OEMVkeys.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Registry</td>
<td>regflush.h</td>
<td></td>
<td>coredll.dll</td>
</tr>
<tr>
<td>Battery Backed RAM</td>
<td>RetentiveMemAPI.h</td>
<td>RetentiveMem.lib</td>
<td>RetentiveMem.dll</td>
</tr>
<tr>
<td>System Timers</td>
<td>usertimers.h</td>
<td>usertimers.lib</td>
<td>usertimers.dll</td>
</tr>
<tr>
<td>Watchdog Timer</td>
<td>watchdog.h</td>
<td>watchdog.lib</td>
<td></td>
</tr>
</tbody>
</table>
The functions described in this section provide application level access to all digital outputs on the RAC6182 via a common interface. The digital outputs include four diagnostic outputs on pins A, B, C and D of header J10 on the system board, and a relay output, with contacts terminated at connector J16, which accessible from the rear of the RAC6182.

These functions allow getting, setting, and toggling of outputs, either individually or simultaneously.

These functions are prototyped in the c header file DiagnosticOutputAPI.h. This file also defines macros for bit masks to be used to access individual outputs. These bit masks may be ORed for simultaneous access of multiple outputs.

**do_ReadPort**

This function reads digital outputs. It is prototyped in DiagnosticOutputAPI.h.

**Syntax**

```c
#include <Windows.h>
#include <DiagnosticOutputAPI.h>

BOOL do_ReadPort(UCHAR *pucData)
```

**Remarks**

pucData is a pointer to a caller allocated UCHAR variable in which current settings of all discrete outputs are to be stored. Settings of individual outputs are accessible via bit masks defined as macros in the header file. These masks are to be applied (separately or bitwise ORed together) to the variable following the call to this function.

Note that the relay contacts are normally closed (i.e., closed when the relay is not energized). Thus, when the relay output is set to 1 or TRUE, the contacts will be open and vice versa.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Digital Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASK_DIAG_PIN_A</td>
<td>J10, pin A (no external access)</td>
</tr>
<tr>
<td>MASK_DIAG_PIN_B</td>
<td>J10, pin B (no external access)</td>
</tr>
<tr>
<td>MASK_DIAG_PIN_C</td>
<td>J10, pin C (no external access)</td>
</tr>
</tbody>
</table>
### Macro Digital Output

<table>
<thead>
<tr>
<th>Macro</th>
<th>Digital Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASK_DIAG_PIN_D</td>
<td>J10, pin D (no external access)</td>
</tr>
<tr>
<td>MASK_RELAY_PIN</td>
<td>J16 (Relay, contacts NC)</td>
</tr>
</tbody>
</table>

**Return Value**

TRUE if read operation was successful, else FALSE.

**Portability**

This function is specific to the RAC6182 hardware.

**Example**

```c
#include <Windows.h>
#include <DiagnosticOutputAPI.h>

int main(void)
{
    UCHAR pucData;
    char buffer[256];

    if (do_ReadPort(&pucData))
        printf("Relay is %s\n", (pucData & MASK_RELAY_PIN) ? "open" : "closed");
    else printf("Error reading digital outputs\n");

    return(0);
}
```

**See Also**

do_WritePort

---

**do_WritePort**

This function writes digital output. It is prototyped in DiagnosticOutputAPI.h.

**Syntax**

```c
#include <Windows.h>
#include <DiagnosticOutputAPI.h>

BOOL do_WritePort(UCHAR ucMask, UCHAR ucData)
```

**Remarks**

ucMask is a bit mask that determines which outputs are modified. If the mask bit for a given output is set to 1, that output will be modified to reflect the corresponding bit in ucData; otherwise the output will not be modified, regardless of the setting of the corresponding bit in ucData.
Note that the relay contacts are normally closed (i.e., closed when the relay is not energized). Thus, when the relay output is set to 1 or TRUE, the contacts will be open and vice versa.

The following macros can be used (separately or bitwise ORed together) to evaluate ucMask.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Digital Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASK_DIAG_PIN_A_J10</td>
<td>pin A (no external access)</td>
</tr>
<tr>
<td>MASK_DIAG_PIN_B_J10</td>
<td>pin B (no external access)</td>
</tr>
<tr>
<td>MASK_DIAG_PIN_C_J10</td>
<td>pin C (no external access)</td>
</tr>
<tr>
<td>MASK_DIAG_PIN_D_J10</td>
<td>pin D (no external access)</td>
</tr>
<tr>
<td>MASK_RELAY_PIN_J16</td>
<td>Relay – contacts NC</td>
</tr>
</tbody>
</table>

**Return Value**

TRUE if write operation was successful, else FALSE.

**Portability**

This function is specific to the RAC6182 hardware.

**Example**

```c
#include <DiagnosticOutputAPI.h>
#include <stdio.h>

int main(void)
{
  UCHAR ucData = 0xff;

  if (do_WritePort(MASK_RELAY_PIN, ucData))
    printf("Relay %s\n", (MASK_RELAY_PIN & ucData) ? "closed" : "opened");
  else printf("Error changing relay state\n");

  return(0);
}
```

**See Also**
do_SetBits, do_ClearBits, do_ToggleBits
do_SetBits

This function sets digital outputs. It is prototyped in DiagnosticOutputAPI.h.

Syntax

```c
#include <Windows.h>
#include <DiagnosticOutputAPI.h>

BOOL do_SetsBits(UCHAR ucMask)
```

Remarks

This function sets any output whose bit is 1 in ucMask to logic TRUE. For the relay output, logic TRUE is equivalent to closed; for TTL outputs, it is equivalent to TTL high level. All pins whose bit is 0 in ucMask are left unchanged.

Note that the relay contacts are normally closed (i.e., closed when the relay is not energized). Thus, when the relay output is set to 1 or TRUE, the contacts will be open and vice versa.

The following macros may be used (separately or bitwise ORed together) to evaluate ucMask:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Digital Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASK_DIAG_PIN_A</td>
<td>J10, pin A (no external access)</td>
</tr>
<tr>
<td>MASK_DIAG_PIN_B</td>
<td>J10, pin B (no external access)</td>
</tr>
<tr>
<td>MASK_DIAG_PIN_C</td>
<td>J10, pin C (no external access)</td>
</tr>
<tr>
<td>MASK_DIAG_PIN_D</td>
<td>J10, pin D (no external access)</td>
</tr>
<tr>
<td>MASK_RELAY_PIN</td>
<td>J16 (Relay – contacts NC)</td>
</tr>
</tbody>
</table>

Return Value

TRUE if read operation was successful, else FALSE.

Portability

This function is specific to the RAC6182 hardware.

See Also

do_ClearBits, do_ToggleBits, do_WritePort
do_ClearBits

This function clears digital outputs. It is prototyped in DiagnosticOutputAPI.h.

**Syntax**

```c
#include <Windows.h>
#include <DiagnosticOutputAPI.h>

BOOL do_ClearBits(UCHAR ucMask)
```

**Remarks**

This function clears any output whose corresponding bit in ucMask is 1 to a logic FALSE. For the relay output, FALSE is equivalent to open; for TTL outputs, it is equivalent to a low level. All outputs whose bits in ucMask are 0 are left unchanged.

Note that the relay contacts are normally closed (i.e., closed when the relay is not energized). Thus, when the relay output is set to 1 or TRUE, the contacts will be open and vice versa.

The following macros may be used (separately or bitwise ORed together) to evaluate ucMask:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Digital Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASK_DIAG_PIN_A J10, pin A (no external access)</td>
<td></td>
</tr>
<tr>
<td>MASK_DIAG_PIN_B J10, pin B (no external access)</td>
<td></td>
</tr>
<tr>
<td>MASK_DIAG_PIN_C J10, pin C (no external access)</td>
<td></td>
</tr>
<tr>
<td>MASK_DIAG_PIN_D J10, pin D (no external access)</td>
<td></td>
</tr>
<tr>
<td>MASK_RELAY_PIN J16 (Relay – contacts NC)</td>
<td></td>
</tr>
</tbody>
</table>

**Return Value**

TRUE if read operation was successful, else FALSE.

**Portability**

This function is specific to the RAC6182 hardware.

**See Also**

do_SetBits, do_ToggleBits, do_WritePort
do_ToggleBits

This function toggles digital outputs. It is prototyped in DiagnosticOutputAPI.h.

Syntax

#include <Windows.h>
#include <DiagnosticOutputAPI.h>

BOOL do_ToggleBits(UCHAR ucMask)

Remarks

Outputs corresponding to bits in ucMask that are set to 1 are toggled. For any output whose bit is 1 in ucMask, if its previous output was TRUE (closed or high) it will be set FALSE (open or low) and vice-versa. All pins whose bit is 0 in ucMask are left unchanged.

Note that the relay contacts are normally closed (i.e., closed when the relay is not energized). Thus, when the relay output is set to 1 or TRUE, the contacts will be open and vice versa.

The following macros may be used (separately or bitwise ORed together) to evaluate ucMask:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Digital Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASK_DIAG_PIN_A</td>
<td>J10, pin A (no external access)</td>
</tr>
<tr>
<td>MASK_DIAG_PIN_B</td>
<td>J10, pin B (no external access)</td>
</tr>
<tr>
<td>MASK_DIAG_PIN_C</td>
<td>J10, pin C (no external access)</td>
</tr>
<tr>
<td>MASK_DIAG_PIN_D</td>
<td>J10, pin D (no external access)</td>
</tr>
<tr>
<td>MASK_RELAY_PIN</td>
<td>J16 (Relay – contacts NC)</td>
</tr>
</tbody>
</table>

Return Value

TRUE if read operation was successful, else FALSE.

Portability

This function is specific to the RAC6182 hardware.

See Also

do_SetBits, do_ClearBits, do_WritePort
The bezel EEPROM provides a total of 256 bytes of non-volatile storage. The first 128 bytes (at offsets 0x00 through 0x7f) are reserved for use by the CE operating system and built-in device drivers (specifically, those drivers that handle devices attached to the bezel, namely the keypad, touch screen and display). The remaining 128 bytes (at offsets 0x80 through 0xff) are reserved for future use.

The functions described in this section can be called by applications to read the system area of the EEPROM (for example, in order to get the keypad identifier, at offset 0x30).

be_GetBezelEEPROMParameter

This function gets EEPROM access mode and CRC status. It is prototyped in bezeeeprom.h.

Syntax

```c
#include <Windows.h>
#include <bezeleeprom.h>

DWORD be_GetBezelEEPROMParameter(DWORD dwParameter, DWORD *pdwData)
```

Remarks

Mode or status information is written to *pdwData, depending on the value of dwParameter. dwParameter may be evaluated with one of the following macros:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEZEL_EEPROM_PARAMETER_USE_CRC</td>
<td>Get the current CRC mode. The value of *pdwData will be 0 if CRC mode is disabled, or 1 if CRC mode is enabled. When CRC mode is enabled, the 16-bit check value stored at offset 0x00 is protected and only the EEPROM contents beginning at offset 0x02 may be read or written. The CRC value at 0x00 is updated when new data are written. Read and write functions return CRC error codes in case of CRC errors. The RAC6182 always boots with CRC mode enabled. When CRC mode is disabled the entire bezel EEPROM is accessible for reading or writing. The first write of new data will invalidate the CRC value at 0x00. However, functions will not return CRC error codes unless CRC mode is again enabled. Thereafter, reads and writes will return CRC error codes until the CRC value is recalculated.</td>
</tr>
<tr>
<td>BEZEL_EEPROM_PARAMETER_CRC_VALID</td>
<td>Get the current state of the CRC check value. The value of *pdwData will be 0 if the 16-bit CRC value stored at offset 0x00 of the EEPROM is invalid, and 1 if it is valid. This is the case whether CRC mode is enabled or disabled.</td>
</tr>
</tbody>
</table>
Return Value

The possible return values are represented by the following macros, defined in bezeleeprom.h:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEZEL_EEPROM_OK</td>
<td>EEPROM present, arguments valid, function completed successfully.</td>
</tr>
<tr>
<td>BEZEL_EEPROM_DEVICE_NOT_PRESENT</td>
<td>No EEPROM detected – either bezel not present or EEPROM on it not functioning.</td>
</tr>
<tr>
<td>BEZEL_EEPROM_INVALID_PARAMETER</td>
<td>Bad parameter passed to function, for example a NULL pointer or an address out of range.</td>
</tr>
<tr>
<td>BEZEL_EEPROM_INVALID_CRC</td>
<td>CRC mode is enabled and the CRC on the EEPROM is currently invalid.</td>
</tr>
</tbody>
</table>

Portability

This function is specific to the RAC6182 hardware.

be_SetBezelEEPROMParameter

This function sets bezel EEPROM access mode or CRC value. It is prototyped in bezeleeprom.h.

Syntax

```c
#include <Windows.h>
#include <bezeleeprom.h>

DWORD be_SetBezelEEPROMParameter(DWORD dwParameter, DWORD *pdwData)
```
**Remarks**

Sets access mode or recalculates CRC, depending on dwParameter and the value of *pdwData. dwParameter may be evaluated using one of the following macros:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEZEL_EEPROM_PARAMETER_USE_CRC</td>
<td>Set CRC mode. If the value of *pdwData is 1, CRC mode is enabled. If the value of *pdwData is 0, CRC mode is disabled. When CRC mode is enabled, the 16-bit check value stored at offset 0x00 is protected and only the EEPROM contents beginning at offset 0x02 may be read or written. The CRC value at 0x00 is updated when new data are written. Read and write functions return CRC error codes in case of CRC errors. The RAC6182 always boots with CRC mode enabled. When CRC mode is disabled the entire bezel EEPROM is accessible for reading or writing. The first write of new data will invalidate the CRC value at 0x00. However, functions will not return CRC error codes unless CRC mode is again enabled. Thereafter, reads and writes will return CRC error codes until the CRC value is recalculated.</td>
</tr>
<tr>
<td>BEZEL_EEPROM_PARAMETER_CRC_VALID</td>
<td>Recalculate the CRC value. If the value of *pdwData is 1, the 16-bit CRC value stored at offset 0x00 of the EEPROM is recalculated. This is the case whether CRC mode is enabled or disabled. If the value of *pdwData is 0, there is no action.</td>
</tr>
</tbody>
</table>

**Return Value**

The possible return values are represented by the following macros, defined in bezeleeprom.h:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEZEL_EEPROM_OK</td>
<td>EEPROM present, arguments valid, function completed successfully.</td>
</tr>
<tr>
<td>BEZEL_EEPROM_DEVICE_NOT_PRESENT</td>
<td>No EEPROM detected – either bezel not present or EEPROM on it not functioning.</td>
</tr>
<tr>
<td>BEZEL_EEPROM_INVALID_PARAMETER</td>
<td>Bad parameter passed to function, for example a NULL pointer or an address out of range.</td>
</tr>
<tr>
<td>BEZEL_EEPROM_INVALID_CRC</td>
<td>CRC mode is enabled and the CRC on the EEPROM is currently invalid.</td>
</tr>
</tbody>
</table>

**Portability**

This function is specific to the RAC6182 hardware.
be_ReadBezelEEPROM

This function reads bezel EEPROM. It is prototyped in bezeleeprom.h.

Syntax

```c
#include <Windows.h>
#include <bezeleeprom.h>

DWORD be_ReadBezelEEPROM(DWORD dwAdress, DWORD dwLength, UCHAR *pucData)
```

Remarks

Reads dwLength bytes starting at offset dwAddress in the EEPROM into a caller allocated buffer beginning at pucData. All reads are implicitly mutexed.

When CRC mode is enabled, EEPROM contents are accessible for reading beginning at offset 0x02. When the CRC is currently invalid, any read will still return the raw data at the requested locations, but will return BEZEL_EEPROM_INVALID_CRC error.

When CRC mode is disabled the entire bezel EEPROM is accessible for reading. Reading any location will never result in the return of a BEZEL_EEPROM_INVALID_CRC error, regardless of the validity of the CRC value.

Return Value

The possible return values are represented by the following macros, defined in bezeleeprom.h:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEZEL_EEPROM_OK</td>
<td>EEPROM present, arguments valid, function completed successfully.</td>
</tr>
<tr>
<td>BEZEL_EEPROM_DEVICE_NOT_PRESENT</td>
<td>No EEPROM detected – either bezel not present or EEPROM on it not functioning.</td>
</tr>
<tr>
<td>BEZEL_EEPROM_INVALID_PARAMETER</td>
<td>Bad parameter passed to function, for example a NULL pointer or an address out of range.</td>
</tr>
<tr>
<td>BEZEL_EEPROM_INVALID_CRC</td>
<td>CRC mode is enabled and the CRC on the EEPROM is currently invalid.</td>
</tr>
</tbody>
</table>

Portability

This function is specific to the RAC6182 hardware.
be_WriteBezelEEPROM

This function (prototyped in bezeleeprom.h) is intended for use by operating system developers only. Rockwell Automation recommends against and does not support use of this function in application programs or user implemented device drivers.

ATTENTION: Improper use of this function could result in disruption of critical system level data.

Function for Watchdog Timer Control

The RAC6182 incorporates a watchdog device to allow automatic reset in case of an application error that involves the application’s loss of control over the hardware. Immediately following system initialization, the watchdog device is disabled. It is enabled when an application issues an initial Watchdog_Tag call. Once enabled, the watchdog device must be tagged periodically by the issuance of additional tags calls to reset the timer in the device; otherwise, the timer will time out. If a timeout occurs, it is assumed that the application or some underlying software has lost control, and the system is reset.

An application can set the time allowed between watchdog tags and can enable or disable the device by calling the function described in this section.

Watchdog_Tag

This function tags the watchdog timer. It is prototyped in watchdog.h.

Syntax

```c
#include <Windows.h>
#include <watchdog.h>

DWORD Watchdog_Tag(DWORD dwTimeout)
```

Remarks

If the value of dwTimeout is 0, the watchdog timeout value is not changed, but if the timer is running, it is reset (tagged).

If the value of dwTimeout is 0xffffffff, the watchdog timer is disabled. If the watchdog is disabled, calling Watchdog_Tag with dwTimeout set to 0 will always return WATCHDOG_TIMEOUT_FAILED as no current watchdog timeout value is defined.

If the value of dwTimeout is other than 0 and 0xffffffff, it is taken to represent the time, in milliseconds, than must elapse before the watchdog will trigger a system reset. If possible, the timer is reset to this value and
started, but if the value of dwTimeout is out of range for the hardware implementation of the timer, the timeout setting of the watchdog is left unmodified and WATCHDOG_TIMEOUT_FAILED is returned. Values of dwTimeout over 5000 should never be used, because they cannot be guaranteed to be within range for a given hardware implementation. At the time of writing, the maximum timeout period supported by the hardware is 3478 msec.

The hardware implementation of the watchdog timer is such that the timer’s precision varies with the timeout value specified. For dwTimeout <= 13, the precision is 1 msec and the accuracy will be within +7/-4% of the specified period. For dwTimeout > 13, the precision is 13.64 msec; that is, actual values are always equal to or within +13.64 msec of the specified values. Accuracy will be within +7/-4% for the actual period, i.e., the specified period rounded up to the next 13.64 msec increment.

**Return Value**

The possible return values are represented by the following macros defined in watchdog.h:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WATCHDOG_OK</td>
<td>Tag or setting of new timeout value succeeded.</td>
</tr>
<tr>
<td>WATCHDOG_NOT_PRESENT</td>
<td>Low-level communication with watchdog device failed.</td>
</tr>
<tr>
<td>WATCHDOG_TIMEOUT_FAILED</td>
<td>Timeout value out of range.</td>
</tr>
</tbody>
</table>

**Portability**

This function is specific to the RAC6182 hardware.

**Functions for Use in Custom Keypad Handlers**

The functions described in this section should never be called by an application program. These are functions that must be implemented in a keypad handler if the handler is to interact properly with the system keypad driver (keypad.dll). The driver is the only software module that should call them.

A keypad handler for the RAC6182 is responsible for mapping virtual key codes supplied by the keypad driver to other virtual key codes, to macro virtual key sequences, etc. and passing the results back to the driver, which sends them on to the main keyboard driver for final processing.

These handler functions are already given a general purpose implementation in Rockwell’s keypad handler (kh.dll), and a default implementation in Rockwell’s handler stub (khstubb.dll).
A custom handler might implement these functions in a different way. However, these functions are subject to redefinition. Accordingly, Rockwell Automation currently recommends against and does not support customer implementations of these functions.

**KhInitialize**

Rockwell Automation currently recommends against and does not support customer implementation of this function.

**KhDeinitialize**

Rockwell Automation currently recommends against and does not support customer implementation of this function.

**KhGetKeyAttributeTable**

Rockwell Automation currently recommends against and does not support customer implementation of this function.

**KhGet GlobalKeySettings**

Rockwell Automation currently recommends against and does not support customer implementation of this function.

**KhSet GlobalKeySettings**

Rockwell Automation currently recommends against and does not support customer implementation of this function.

**KhTranslateVKey**

Rockwell Automation currently recommends against and does not support customer implementation of this function.
Streams Interface for Keypad Driver Control

The keypad related macros and special IOCTLs defined in KeypadAPI.h are intended for use by system developers only. Rockwell recommends against and does not support their use by customers.

Standard I/O streams functions called with these special constants as arguments provide a means for the calling software module to control the system keypad driver (keypad.dll). The software module that utilizes this means is the system keypad handler (kh.dll).

Streams Interface for Touchscreen Control

The macros and special IOCTLs defined in atmelapi.h are intended for use by system developers only. Rockwell recommends against and does not support their use by customers.

Standard I/O streams functions called with these special constants as arguments provide a means for the calling software module to control the system touch screen controller. The software module that utilizes this means is the system control panel.

Functions for LED Control

Three LEDs are available on the RAC6182 and may be accessed by applications.

LED 0 – Warning LED. This LED is toggled on and off during the boot process and is set to off at the end of boot. After boot it is not used by any operating system components and is available for use by an application to signal warning conditions.

LED 1 – NUMLOCK LED. The keypad driver in the operating system controls this LED. When first loaded, the driver assumes the default state of NUMLOCK on and accordingly turns this LED on. The driver will set or clear this LED upon change of the keypad NUMLOCK state. This LED is application accessible, but its use by applications is not recommended except the RAC6182 is configured without a keypad.

LED 2 – Power LED. This LED is toggled on and off during the boot process and is left on at the end of boot. The operating system does not use this LED after boot. This LED is application accessible, but its use by applications is not recommended except possibly in a power fail/brownout condition.
NLedGetDeviceInfo

This function gets the LED Status Information.

Syntax

#include <Windows.h>
#include <nledapi.h>

BOOL NLedGetDeviceInfo(UINT nInfoId, void *pOutput)

Remarks

Use this function to request information about the system LEDs.

Information that is accessible with this function includes the number of LEDs installed, the capabilities of each installed LED, and the current settings for each installed LED. In general, an LED may be in one of three states: on, off, or blinking. The on and off states can be obtained without any special ado, but the blinking state can be obtained only by specifying the values of adjustable parameters that determine the blink rate and duty cycle. Thus, the capabilities information accessible with this function relates to the adjustability of blink parameters. The settings information relates to the basic state and to the additional settings that determine blink rate and duty cycle.

Note that for blink to be possible, at least two of the following parameters must be adjustable: on time, off time, total cycle time (the sum of on and off times.) As of this writing, the capabilities of the RAC6182 LEDs include adjustability of only one of these parameters. Therefore, intrinsic blink is not available. However, it is still possible to make the LEDs blink using NLedSedDevice to toggle between the “on” and “off” states at an interval determined by a separate timer such as is available with the RAC6182 user timer functions.

Note: Do not confuse this function with NLedDriverGetDeviceInfo() described in the Microsoft CE documentation, which is a kernel level function, not callable by applications.

nInfoId may be evaluated with one of the following macros defined in nleddrv.h (included by nledapi.h):

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
</table>
| NLED_COUNT_INFO_ID           | Use to request the number of LEDs installed. pOutput should point to a caller allocated structure defined (in nleddrv.h) as follows:  
<p>|                              | struct NLED_COUNT_INFO {                                                    |
|                              |   UINT cLeds; // Count of LEDs                                              |
|                              | };                                                                          |</p>
<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NLED_SUPPORTS_INFO_ID</td>
<td>Use to request capability information about any one of the LEDs in the system.</td>
</tr>
<tr>
<td></td>
<td>pOutput should point to a caller allocated structure defined (in nleddrv.h) as follows:</td>
</tr>
<tr>
<td></td>
<td>struct NLED_SUPPORTS_INFO {</td>
</tr>
<tr>
<td></td>
<td>UINT LedNum;</td>
</tr>
<tr>
<td></td>
<td>LONG lCycleAdjust; // Granularity (usec)</td>
</tr>
<tr>
<td></td>
<td>BOOL fAdjustTotalCycleTime;</td>
</tr>
<tr>
<td></td>
<td>BOOL fAdjustOnTime;</td>
</tr>
<tr>
<td></td>
<td>BOOL fAdjustOffTime;</td>
</tr>
<tr>
<td></td>
<td>BOOL fMetaCycleOn;</td>
</tr>
<tr>
<td></td>
<td>BOOL fMetaCycleOff;</td>
</tr>
<tr>
<td></td>
<td>};</td>
</tr>
<tr>
<td></td>
<td>Prior to calling this function, the individual LED for which information is sought must be selected by evaluating LedNum using one of the following macros (defined in nledapi.h):</td>
</tr>
<tr>
<td></td>
<td>ID_WARNING_LED</td>
</tr>
<tr>
<td></td>
<td>ID_NUMLOCK_LED</td>
</tr>
<tr>
<td></td>
<td>ID_POWER_LED</td>
</tr>
<tr>
<td></td>
<td>When the function returns, the other members of the structure will contain information about the LED’s capabilities. For Booleans, TRUE means that the parameter is adjustable with a NLedSetDevice call. lCycleAdjust is the resolution of the timer (in usec) that controls blink-on and blink-off.</td>
</tr>
<tr>
<td>NLED_SETTINGS_INFO_ID</td>
<td>Use to request current settings of any one of the LEDs in the system.</td>
</tr>
<tr>
<td></td>
<td>pOutput should point to a caller allocated structure defined (in nleddrv.h) as follows:</td>
</tr>
<tr>
<td></td>
<td>struct NLED_SETTINGS_INFO {</td>
</tr>
<tr>
<td></td>
<td>UINT LedNum;</td>
</tr>
<tr>
<td></td>
<td>INT OffOnBlink; // 0=off, 1=on, 2=blink</td>
</tr>
<tr>
<td></td>
<td>LONG TotalCycleTime; // (usec)</td>
</tr>
<tr>
<td></td>
<td>LONG OnTime; // blink-on time(usec)</td>
</tr>
<tr>
<td></td>
<td>LONG OffTime; // blink-off time(usec)</td>
</tr>
<tr>
<td></td>
<td>INT MetaCycleOn; // num blink-on cycles</td>
</tr>
<tr>
<td></td>
<td>INT MetaCycleOff; // num blink-off cycles</td>
</tr>
<tr>
<td></td>
<td>};</td>
</tr>
<tr>
<td></td>
<td>Prior to calling this function, the LED for which setting information is sought must be selected. This is done by evaluating LedNum using one of the following macros (defined in nledapi.h):</td>
</tr>
<tr>
<td></td>
<td>ID_WARNING_LED</td>
</tr>
<tr>
<td></td>
<td>ID_NUMLOCK_LED</td>
</tr>
<tr>
<td></td>
<td>ID_POWER_LED</td>
</tr>
<tr>
<td></td>
<td>When the function returns, current settings will be stored in the corresponding members of the NLED_SETTINGS_INFO structure.</td>
</tr>
</tbody>
</table>

**Return Value**

One of the following: TRUE or FALSE.
Portability
The arguments to this function are specific to the RAC6182 hardware.

NledSetDevice
This function sets the LEDs.

Syntax
#include <Windows.h>
#include <nledapi.h>

BOOL NLedSetDevice(UINT nDeviceId, void *pInput)
Remarks

Use this function to set the operating states of the system LEDs.

Note: Before calling this function, it is a good idea to issue a call to the NLedGetDeviceInfo function to verify the presence of the LED of interest and to get its capabilities. As of this writing, RAC6182s with integrated LCDs are equipped with three front panel LEDs which have only “on” and “off” capabilities.

Note: Do not confuse this function with NLedDriverSetDevice() described in the Microsoft CE documentation, which is a kernel level function, not callable by applications.

nInfoId must be evaluated with the NLED_SETTINGS_INFO_ID macro defined in nleddrv.h (included by nledapi.h).

pInput must point to a caller allocated structure, defined in nleddrv.h as follows:

```c
struct NLED_SETTINGS_INFO {
    UINT LedNum;
    INT OffOnBlink; // 0=off, 1=on, 2=blink
    LONG TotalCycleTime; // (usec)
    LONG OnTime; // blink-on time(usec)
    LONG OffTime; // blink-off time(usec)
    INT MetaCycleOn; // num blink-on cycles
    INT MetaCycleOff; // numblink-off cycles
};
```

Prior to calling this function, the LED that is to be set must be selected. This is done by evaluating LedNum using one of the following macros (defined in nledapi.h):

- ID_WARNING_LED
- ID_NUMLOCK_LED
- ID_POWER_LED

Other parameters should be set, based on capabilities for the LED in question obtained with an NLedGedDeviceInfo call.

Return Value

One of the following: TRUE or FALSE.

Portability

The arguments to this function are specific to the RAC6182 hardware.

Functions for Use in PCI Device Drivers

The functions necessary to access PCI configuration space, map PCI memory or IO space, and handle interrupts on the PCI card are HAL (Hardware Abstraction Layer) calls and several other CE-specific Win32 calls. The basic operation of these calls is already documented by
Microsoft, so this document will focus just on RAC6182 specific features or limitations and recommended usage in application level code or user mode device drivers.

ceddk.h and ceddk.lib must be used for compilation and linking in addition to the default libraries.
HalTranslateBusAddress

This function translates the PCI Bus Address.

Syntax
#include <Windows.h>
#include <ceddk.h>

BOOL HalTranslateBusAddress(
    INTERFACE_TYPE InterfaceType,
    ULONG BusNumber,
    PHYSICAL_ADDRESS BusAddress,
    PULONG AddressSpace,
    PHYSICAL_ADDRESS TranslatedAddress
)

Remarks
HalTranslateBusAddress() converts a PCI memory or IO space address into a physical address which may be used to map virtual address space to the memory or IO using MmMapIoSpace().

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InterfaceType</td>
<td>PCIBus is the only allowed</td>
</tr>
<tr>
<td>BusNumber</td>
<td>The RAC6182 PCI slot is on bus 1.</td>
</tr>
<tr>
<td>BusAddress</td>
<td>must have its upper 32 bits equal to 0, and lower 32 bits the PCI memory or IO space address obtained from the PCI address space registers. Values from PCI address space registers should have the non-address bits such as I/O, prefetch, etc. masked off before being put in the lower part of BusAddress</td>
</tr>
<tr>
<td>AddressSpace</td>
<td>set to 0 for memory, 1 for IO.</td>
</tr>
<tr>
<td>Translated Address</td>
<td>If parameters are valid, the function will return with a value usable by MmMapIoSpace().</td>
</tr>
</tbody>
</table>

Return Value
One of the following: TRUE or FALSE.

Portability
The arguments to this function are specific to the RAC6182 hardware.
HalGetBusDataByOffset

This function gets the PCI Bus Data by offset.

Syntax
#include <Windows.h>
#include <ceddk.h>

ULONG HalGetBusDataByOffset(
    BUS_DATA_TYPE BusDataType,
    ULONG BusNumber,
    ULONG SlotNumber,
    PVOID Buffer,
    ULONG Offset,
    ULONG Length
)

Remarks
HalGetBusDataByOffset() retrieves PCI configuration space information, such as addresses and interrupt information, for a PCI device.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BusDataType</td>
<td>PCIConfiguration</td>
</tr>
<tr>
<td>BusNumber</td>
<td>The RAC6182 PCI slot is on bus 1.</td>
</tr>
<tr>
<td>SlotNumber</td>
<td></td>
</tr>
<tr>
<td>Buffer</td>
<td></td>
</tr>
<tr>
<td>Offset</td>
<td></td>
</tr>
<tr>
<td>Length</td>
<td></td>
</tr>
</tbody>
</table>

Return Value

Portability
The arguments to this function are specific to the RAC6182 hardware.
HalSetBusDataByOffset

This function sets the PCI Bus Data by offset.

Syntax
#include <Windows.h>
#include <ceddk.h>

ULONG HalSetBusDataByOffset(
    BUS_DATA_TYPE BusDataType,
    ULONG BusNumber,
    ULONG SlotNumber,
    PVOID Buffer,
    ULONG Offset,
    ULONG Length
)

Remarks
HalSetBusDataByOffset() sets PCI configuration space information. This function should be used only to set device-specific configuration information.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BusDataType</td>
<td>PCIConfiguration</td>
</tr>
<tr>
<td>BusNumber</td>
<td>The RAC6182 PCI slot is on bus 1.</td>
</tr>
<tr>
<td>SlotNumber</td>
<td>The RAC6182 PCI slot number is 1.</td>
</tr>
<tr>
<td>Buffer</td>
<td></td>
</tr>
<tr>
<td>Offset</td>
<td></td>
</tr>
<tr>
<td>Length</td>
<td></td>
</tr>
</tbody>
</table>

Return Value
One of the following:

Portability
The arguments to this function are specific to the RAC6182 hardware.
**MmMapIoSpace**

This function maps the PCI IO space.

**Syntax**

```c
#include <Windows.h>
#include <ceddk.h>

PVOID MmMapIoSpace(
    PHYSICAL_ADDRESS PhysicalAddress,
    ULONG NumberOfBytes,
    BOOLEAN CacheEnable
)
```

**Remarks**

MmMapIoSpace() maps a physical address range into a virtual address range usable by an application. This function should be used on PCI memory or IO range information obtained with HalGetBusDataByOffset() and translated with HalTranslateBusAddress().

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PhysicalAddress</td>
<td></td>
</tr>
<tr>
<td>NumberOfBytes</td>
<td></td>
</tr>
<tr>
<td>CacheEnable</td>
<td></td>
</tr>
</tbody>
</table>

**InterruptInitialize**

This function initializes the PCI interrupt.

**Syntax**

```c
#include <Windows.h>
#include <ceddk.h>

BOOL InterruptInitialize(
    DWORD idInt,
    HANDLE hEvent,
    LPVOID pvData,
    DWORD cbData
)
```

**Remarks**

InterruptInitialize() associates a virtual interrupt number with an application created event so that the application can use WaitFor*() functions to wait for the interrupt to occur. For RAC6182, the virtual interrupt is obtained from the PCI configuration space interrupt register.
(offset 0x0C) using HalGetBusDataByOffset(). The PCI slot interrupt is disabled until this function is called.

**Portability**
The argument to this function is specific to the RAC6182 hardware.

### InterruptDisable

This function disables the PCI interrupt.

**Syntax**

```c
#include <Windows.h>
#include <ceddk.h>

VOID InterruptDisable ( DWORD idInt )
```

**Remarks**

InterruptDisable() disables the virtual interrupt.

**Return Value**

One of the following:

**Portability**
The argument to this function is specific to the RAC6182 hardware.

### InterruptDone

This function cleans up after the PCI interrupt.

**Syntax**

```c
#include <Windows.h>
#include <ceddk.h>

VOID InterruptDone ( DWORD idInt )
```

**Remarks**

InterruptDone() signals that the application which registered for the virtual interrupt using InterruptInitialize() is done handling the interrupt. The interrupt is reenabled when InterruptDone() is called.

**Return Value**

One of the following:
Sample Code for a Simple PCI Slot Device

Portability

The argument to this function is specific to the RAC6182 hardware.

This code assumes a simple device with one memory address space size 4K configured at PCI configuration offset 0x10 and one IO address space size 16K configured at PCI configuration offset 0x14. It has a power management register at offset 0x40 that must be set to bring the device out of power down mode. The device periodically interrupts us to print data or tell us it has died.

```c
#include <windows.h>
#include <ceddk.h>

// values currently defined for RAC6182.
// these will be provided in a pcislot.h file
#define PCI_SLOT_BUS_NUMBER 1
#define PCI_SLOT_DEVICE_NUMBER 1

// bogus items for our simple card
#define CARD_VENDOR_ID 0x1234
#define CARD_DEVICE_ID 0x5678
#define CARD_POWER_MGMT_ON 0x00000001
#define CARD_MEMORY_SIZE (4*1024)
#define CARD_IO_SIZE (16*1024)
#define CARD_SIGNAL_DEAD 0xDEAD

int main(void)
{
    PCI_COMMON_CONFIG PCIConfig;
    HANDLE waitevent;
    DWORD interrupt;
    PHYSICAL_ADDRESS pa,pa2;
    volatile UCHAR *IOSpace;
    volatile UCHAR *MemSpace;
    ULONG value;

    NKDbgPrintfW(TEXT("PCI card sample\n");
    if(HalGetBusDataByOffset(PCIConfiguration,PCI_SLOT_BUS_NUMBER,
        PCI_SLOT_DEVICE_NUMBER,&PCIConfig,0,sizeof(PCIConfig)) !=
        sizeof(PCIConfig)) {
        NKDbgPrintfW(TEXT("failed to get PCI slot config info\n");
        return(-1);
    }

    // check that our card is present
    if(PCIConfig.VendorID!=CARD_VENDOR_ID ||
        PCIConfig.DeviceID!=CARD_DEVICE_ID) {
        NKDbgPrintfW(TEXT("vendor %04x device %04x not our card\n"),
            PCIConfig.VendorID,PCIConfig.DeviceID);
        return(-1);
    }

    // power on to our bogus card to demonstrate a device-specific config
    // space write
    value=CARD_POWER_MGMT_ON;
    if(HalSetBusDataByOffset(PCIConfiguration,PCI_SLOT_BUS_NUMBER,
        PCI_SLOT_DEVICE_NUMBER,&value,0x40,sizeof(value)) !=
        sizeof(value)) {
        NKDbgPrintfW(TEXT("failed to set PCI config info\n");
        return(-1);
    }

    return(0);
}
```

Publication 6182-UM002B-EN-P
NKDbgPrintfW(TEXT("failed to set device specific PCI config
value\n"));
    return(-1);
}
// get virtual interrupt #
interrupt=PCICfg.u.type0.InterruptLine;
// can use manual reset or named event if desired
waitevent=CreateEvent(NULL,FALSE,FALSE,NULL);
if(!waitevent || waitevent==INVALID_HANDLE_VALUE) {
    NKDbgPrintfW(TEXT("failed createevent()\n"));
    return(-1);
}
if(!InterruptInitialize(interrupt,waitevent,NULL,0)) {
    NKDbgPrintfW(TEXT("failed claiming interrupt %d\n"),interrupt);
    return(-1);
}
// handle 1st address space – 4K memory
pa.HighPart=0;
pa.LowPart=PCICfg.u.type0.BaseAddresses[0]&
PCI_ADDRESS_MEMORY_ADDRESS_MASK);
value=0;
if(!HalTranslateBusAddress(PCIBus,PCI_SLOT_BUS_NUMBER,pa,&value,&pa2)) {
    NKDbgPrintfW(TEXT("failed translating mem address\n"));
    return(-1);
}
if(!(MemSpace=MmMapIoSpace(pa2,CARD_MEMORY_SIZE,FALSE))) {
    NKDbgPrintfW(TEXT("failed mapping mem address\n"));
    return(-1);
}
// handle 2nd address space – 16K IO
pa.HighPart=0;
pa.LowPart=PCICfg.u.type0.BaseAddresses[1]&
PCI_ADDRESS_IO_ADDRESS_MASK);
value=1;
if(!HalTranslateBusAddress(PCIBus,PCI_SLOT_BUS_NUMBER,pa,&value,&pa2)) {
    NKDbgPrintfW(TEXT("failed translating io address\n"));
    return(-1);
}
if(!(IOSpace=MmMapIoSpace(pa2,CARD_MEMORY_SIZE,FALSE))) {
    NKDbgPrintfW(TEXT("failed mapping IO address\n"));
    return(-1);
}
// everything mapped and initialized OK, now we become an interrupt
handler
while(1) {
    value=WaitForSingleObject(waitevent,1000);
    if(value==WAIT_TIMEOUT) {
        NKDbgPrintfW(TEXT("<yawn!>\n"));
        // do not InterruptDone() on a time out or if event other
        // than the interrupt-associated event triggered
    } else if(value==WAIT_OBJECT_0) {
        // interrupt occurred on PCI slot device
        if(*((ULONG *)MemSpace)==CARD_SIGNAL_DEAD ||
            *(ULONG *)IOSpace==CARD_SIGNAL_DEAD) {
            NKDbgPrintfW(TEXT("card dead\n"));
            InterruptDisable(interrupt);
The functions prototyped in osupdateapi.h are intended for use by system developers only. Rockwell Automation recommends against and does not support use of these functions in applications and device drivers.

Methods for updating the RAC6182 operating system software are discussed in the RAC6182 User’s Manual.

**osu_UpdateOSFromFile**

Rockwell Automation recommends against and does not support use of this function.

**osu_RemoteUpdateOSFromRAM**

Rockwell Automation recommends against and does not support use of this function.

A RAC6182 has a Windows CE registry which is stored in the RAM of the device. Since RAM data are valid only while the RAC6182 is powered on, a persistent backup of the registry is maintained on the Disk-On-Chip.

The Windows CE operating system does not automatically flush the registry in RAM to persistent storage, therefore if registry settings are changed by an application, the application should invoke a FlushRegistry operation to ensure that the settings will persist from one operating session to another.

**Note:** Because of the relatively large amount of time required to flush the registry to flash, it is highly recommended that applications adding or changing registry information complete a set of changes before issuing a flush rather than attempting to flush after every single update.
FlushRegistry

This function is the Flush Registry. It is prototyped in regflush.h.

Syntax

#include <Windows.h>
#include <regflush.h>

BOOL RegistryFlush(void)

Remarks

This function is defined as a macro in regflush.h. This function commands the operating system to flush the entire registry to the persistent registry storage. The procedure that occurs when this API is invoked is as follows:

1. The entire registry information is collected from Windows CE.
2. This information is compressed to save space, and also results in a time savings when going to flash memory such as the Disk-On-Chip.
3. Any existing temporary persistent registry file is deleted.
4. The temporary persistent registry file is created and the compressed registry information written out.
5. Any existing backup persistent registry file is deleted.
6. Any existing primary persistent registry file is moved to the backup persistent registry file.
7. The temporary persistent registry file is moved to the primary persistent registry file and has its attributes set to read-only, hidden, and system.

The function does not return unless there is an error or the flush is successfully completed. Any fatal errors such as failing to allocate enough working RAM, failing to create or write the temporary persistent registry file or failure to move this file to be the primary persistent registry file will result in failure of the function. File system errors will result if the Disk-On-Chip is not properly formatted, has been corrupted, or does not have enough space free.

The registry flush procedure requires 512K+2*(compressed registry size) bytes of RAM be free as working space for storing and compressing the registry. The Disk-On-Chip file-system needs to have (compressed registry size rounded up to nearest cluster size) bytes free for registry flush to succeed. Typical registry size for the base operating system are on the order of 128K-160K uncompressed which compresses very well to 32K-40K. Registry keys added by applications will certainly increase
the overall size and the nature of the content may also affect compressibility.

Time involved to flush the registry varies based on size of the registry, and can also vary based on state of the Disk-On-Chip (e.g. if flash sectors need erased and written this takes significantly longer than just writing to previously erased flash sectors). For the registry in the base operating system time to flush has been seen to vary between 100 to 500 ms. The time to write to the Disk-On-Chip is by far the dominant factor; reading the registry from CE into working RAM and compressing it typically takes less than 10-20 ms. of this time. Larger registry sizes due to key additions by applications can be expected to vary in a higher range of time for a registry flush to complete.

Return Value
TRUE if successful, otherwise FALSE

Portability
This function is specific to the RAC6182 hardware.

Function to Adjust Allocation of DRAM

SetSystemMemoryDivision

This function sets the amount of DRAM allocated to system. It is prototyped in othersdk.h.

Syntax
#include <Windows.h>
#include <othersdk.h>

DWORD SetSystemMemoryDivision(DWORD dwStorePages)

Remarks
This function is called by an application to set the amount of DRAM allocated to the system.

Total DRAM installed can be obtained with a call to rm_GetParameter(), using RM_PARAMETER_PHYSICAL_MEMORY_SIZE as the first argument (see description below). This memory is divided into two logical partitions, one for the Object Store (RAMDISK), and one for system memory. The memory available for the Object Store will be the total amount of memory less the amount allocated to the system.
dwStorePages the number of 4KB pages to be allocated to the system.

**Important**: Windows CE V2.12 could exhibit problems if more than 32MB of DRAM is allocated to the system.

**Return Value**
TRUE if successful, otherwise FALSE.

**Portability**
This function is specific to the RAC6182 hardware.

**Functions to Get/Set Misc Parameters**
The functions described here may be called by an application program to get or set the values of certain system parameters.

The following table enumerates currently defined parameters that can be accessed with these functions. Type of data, minimum size, and whether set, get, or both are allowed are given. This table may be expanded in the future to add new parameter types without adding new functions.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Macro Identifier</th>
<th>Get or Set</th>
<th>Type</th>
<th>Size (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical Memory Size</td>
<td>RM_PARAMETER_PHYSICAL_MEMORY_SIZE</td>
<td>Get</td>
<td>DWORD</td>
<td>4</td>
</tr>
<tr>
<td>Pointer to Cached Physical Memory</td>
<td>RM_PARAMETER_PHYSICAL_MEMORY_CACHED_POINTER</td>
<td>Get</td>
<td>void *</td>
<td>4</td>
</tr>
<tr>
<td>Pointer to Uncached Physical Memory</td>
<td>RM_PARAMETER_PHYSICAL_MEMORY_UNCACHED_POINTER</td>
<td>Get</td>
<td>void *</td>
<td>4</td>
</tr>
<tr>
<td>CPU Speed (Hz)</td>
<td>RM_PARAMETER_CPU_SPEED_HZ</td>
<td>Get</td>
<td>DWORD</td>
<td>4</td>
</tr>
<tr>
<td>Windows CE Version</td>
<td>RM_PARAMETER_WINDOWS_CE_VERSION</td>
<td>Get</td>
<td>WCHAR</td>
<td>80 x 2</td>
</tr>
<tr>
<td>Operating System Boot Image Version</td>
<td>RM_PARAMETER_OS_FIRMWARE_VERSION</td>
<td>Get</td>
<td>WCHAR</td>
<td>80 x 2</td>
</tr>
<tr>
<td>Boot ROM Firmware Version</td>
<td>RM_PARAMETER_BOOTROM_FIRMWARE_VERSION</td>
<td>Get</td>
<td>WCHAR</td>
<td>80 x 2</td>
</tr>
<tr>
<td>Aux Microcontroller Firmware Version</td>
<td>RM_PARAMETER_MICROCONTROLLER_FIRMWARE_VERSION</td>
<td>Get</td>
<td>WCHAR</td>
<td>80 x 2</td>
</tr>
<tr>
<td>Parameter</td>
<td>Macro Identifier</td>
<td>Get or Set</td>
<td>Type</td>
<td>Size (Bytes)</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>-----------------------------------------------------------------------</td>
<td>------------</td>
<td>----------</td>
<td>--------------</td>
</tr>
<tr>
<td>Debug output on COM2</td>
<td>RM_PARAMETER_ENABLE_SERIAL_DEBUG_ON_BOOT</td>
<td>Both</td>
<td>BOOL</td>
<td>4</td>
</tr>
<tr>
<td>LCD Brightness</td>
<td>RM_PARAMETER_LCD_BRIGHTNESS</td>
<td>Both</td>
<td>DWORD</td>
<td>4</td>
</tr>
<tr>
<td>LCD Contrast</td>
<td>RM_PARAMETER_LCD_CONTRAST</td>
<td>Both</td>
<td>DWORD</td>
<td>4</td>
</tr>
<tr>
<td>MAC Addr of on board Ethernet</td>
<td>RM_PARAMETER_ONBOARD_ETHERNET_MAC_ADDRESS</td>
<td>Get</td>
<td>UCHAR</td>
<td>6</td>
</tr>
<tr>
<td>Cursor Status</td>
<td>RM_PARAMETER_CURSOR_ENABLED</td>
<td>Both</td>
<td>BOOL</td>
<td>4</td>
</tr>
<tr>
<td>Debug output on COM2</td>
<td>RM_PARAMETER_ENABLE_SERIAL_DEBUG</td>
<td>Both</td>
<td>BOOL</td>
<td>4</td>
</tr>
<tr>
<td>Memory pointer</td>
<td>RM_PARAMETER_PHYSICAL_ADDRESS (CE V3.0 SDK only)</td>
<td>Get</td>
<td>void *</td>
<td>4</td>
</tr>
</tbody>
</table>

**GetParameter**

This function gets parameters. It is prototyped in miscsystem.h.

**Syntax**

```c
#include <Windows.h>
#include <miscsystem.h>

DWORD rm_GetParameter(DWORD dwParameter, DWORD *dwSize, VOID *pvData)
```

**Remarks**

dwParameter may be evaluated with any one of the following macros defined in miscsystem.h in order to select the system parameter value to be retrieved:

- RM_PARAMETER_PHYSICAL_MEMORY_SIZE
- RM_PARAMETER_PHYSICAL_MEMORY_CACHED_POINTER
- RM_PARAMETER_PHYSICAL_MEMORY_UNCACHED_POINTER
- RM_PARAMETER_CPU_SPEED_HZ
- RM_PARAMETER_WINDOWS_CE_VERSION
- RM_PARAMETER_OS_FIRMWARE_VERSION
- RM_PARAMETER_BOOTROM_FIRMWARE_VERSION
- RM_PARAMETER_MICROCONTROLLER_FIRMWARE_VERSION
- RM_PARAMETER_ENABLE_SERIAL_DEBUG_ON_BOOT
- RM_PARAMETER_LCD_BRIGHTNESS
RM_PARAMETER_LCD_CONTRAST
RM_PARAMETER_ONBOARDETHERNET_MAC_ADDRESS
RM_PARAMETER_CURSOR_ENABLED
RM_PARAMETERENABLE_SERIAL_DEBUG
RM_PARAMETER_PHYSICAL_ADDRESS (CE V3.0 only)

dwSize is a pointer to a caller allocated DWORD whose value will
represent the number of bytes at pvData.

pvData is a pointer to a caller allocated buffer which will contain
information related to the current settings for the selected parameter.
The buffer must be large enough to contain the information requested
and must be aligned as required. For example, if a request for a
parameter will result in *pvData being filled with a DWORD value,
*pvData must be DWORD aligned. Please refer to the table in the
introduction to this section of the manual for the data types associated
with the various readable parameters.

Return Value
One of the following:

RM_ERROR_OK - Parameter valid, size large enough, succeeded
RM_ERRORINVALID_PARAMETER - Bad dwParameter or
NULL dwSize or pvData
RM_ERRORINVALID_BUFFER_SIZE - Buffer size too small for
requested parameter

Portability
The arguments to this function are specific to the RAC6182 hardware.

SetParameter

This function sets parameters. It is prototyped in micsystem.h.

Syntax

#include <Windows.h>
#include <micsystem.h>

DWORD rm_SetParameter(DWORD dwParameter, DWORD *dwSize, VOID *
*pvData)

Remarks

dwParameter may be evaluated with any one of the following macros
defined in micsystem.h in order to select the system parameter to be set:
RM_PARAMETER_ENABLE_SERIAL_DEBUG_ON_BOOT
RM_PARAMETER_LCD_BRIGHTNESS
RM_PARAMETER_LCD_CONTRAST
RM_PARAMETER_CURSOR_ENABLED
RM_PARAMETER_ENABLE_SERIAL_DEBUG
RM_PARAMETER_PHYSICAL_ADDRESS (CE V3.0 only)

dwSize is a pointer to a caller allocated DWORD. *dwSize should be evaluated with sizeof(<type_of_parameter>) or with the size in bytes of the parameter type as indicated in the table at the beginning of this section. When the function returns, the value of *dwSize will indicates the actual number of bytes from pvData used to set the selected parameter.

pvData is a pointer to a caller allocated buffer containing the setting data to be applied to the selected parameter. The buffer must be sized and aligned according to the type of the data. For example, if *pvData will be filled with a DWORD value, it must be DWORD aligned. Please refer to the table in the introduction to this section of the manual for details.

Return Value
One of the following:

RM_ERROR_OK - Parameter valid, size large enough, succeeded
RM_ERROR_INVALID_PARAMETER - Bad dwParameter or NULL dwSize or pvData
RM_ERROR_INVALID_BUFFER_SIZE - Buffer size too small for requested parameter

Portability
The arguments to this function are specific to the RAC6182 hardware.

Functions for Accessing System Timers

The hardware of the RAC6182 has provides a number of timers of varying precision, flexibility, and range. Some of these timers may be used for other operating-system level purposes such as reschedule timer interrupts, PWM for LCD panels, RS485 implementation, etc. However, at least one of these timers is guaranteed to be available for general purpose use to application programs.

Since the number of timers is only guaranteed to be at least one and RAC6182 supports the possibility of multiple applications executing, applications should use the timer APIs along with a fallback to less precise Sleep() or busy-wait timing if timers are not available for their usage.
**UserTimerGetNumberOfTimers**

This function gets the number of available timers. It is prototyped in usertimers.h.

**Syntax**

```c
#include <Windows.h>
#include <usertimers.h>

DWORD UserTimerGetNumberOfTimers(void)
```

**Remarks**

Returns total number of application accessible timers on the system.

**Return Value**

Total number of application accessible timers available.

**Portability**

This function is specific to the RAC6182 hardware.

**See Also**

UserTimerClaim

---

**UserTimerClaim**

This function claims, or releases, access to user timers. It is prototyped in usertimers.h.

**Syntax**

```c
#include <Windows.h>
#include <usertimers.h>

DWORD UserTimerClaim(DWORD TimerNumber,BOOL Claim)
```
Remarks

This function is used to claim or release exclusive access to a specific timer. A timer must be claimed before any function taking a TimerNumber as a parameter can be used. A timer must be released to allow any other application to claim and use the timer.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimerNumber</td>
<td>TimerNumber is 0 based (i.e. if 2 timers are present on the system, they are timer #0 and timer #1).</td>
</tr>
<tr>
<td>Claim</td>
<td>Claim is TRUE to claim access to a timer, FALSE to release it.</td>
</tr>
</tbody>
</table>

Return Value

Possible return values are represented by macros defined in usertimers.h:

- USER_TIMER_OK - Successfully claimed or released timer
- USER_TIMER_INVALID_TIMER - Timer number not present on system
- USER_TIMER_NOT_CLAIMED - Another application already claimed the timer so this application could not claim it
- USER_TIMER_ALREADYCLAIMED - This application has already claimed this timer

Portability

This function is specific to the RAC6182 hardware.

See Also

UserTimerGetNumberOfTimers

UserTimerRequestFrequency

This function sets the frequency of user timers. It is prototyped in usertimers.h.

Syntax

```c
#include <Windows.h>
#include <usertimers.h>

DWORD UserTimerRequestFrequency(DWORD TimerNumber, DWORD *Frequency)
```

Remarks

Requests that the timer use a frequency as close as possible to a specified frequency for its count. Hardware timer capabilities vary significantly, so there may only be one or certain gradations of real frequencies possible.
Applications must check the *Frequency returned and use it in their counter calculations.

**Note:** As of this writing, RAC6182 user timers operate at a fixed frequency of 75MHz. Thus, attempts to adjust the timers to frequencies other than 75MHz will not be effective. However, the timers’ frequencies may become adjustable in future releases of the RAC6182, and no guarantee can be made that the timers will always have a fundamental frequency of 75MHz.

---

**ATTENTION:** This function has unpredictable results if called when the timer is running.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimerNumber</td>
<td>Number of a timer previously claimed.</td>
</tr>
<tr>
<td>Frequency</td>
<td>Pointer to an application allocated DWORD containing the frequency (in Hz) to which the timer is to be set. When the function is successful, it changes *Frequency to the actual value used.</td>
</tr>
</tbody>
</table>

**Return Value**

Possible return values are represented by macros defined in usertimers.h:

- USER_TIMER_OK - Successfully claimed or released timer
- USER_TIMER_INVALID_TIMER - Timer number not present on system
- USER_TIMER_NOTCLAIMED - Another application already has claimed the timer so this application could not claim it
- USER_TIMER_INVALID_PARAMETER - Invalid value

**Portability**

This function is specific to the RAC6182 hardware.

**See Also**

UserTimerClaim and UserTimerSet
UserTimerSet

This function sets the count of user timer and start timing. It is prototyped in usertimers.h.

Syntax

```c
#include <Windows.h>
#include <usertimers.h>

DWORD UserTimerSet(DWORD TimerNumber, DWORD Count)
```

Remarks

Sets the timer to a given countdown value and starts the timer. The frequency of the count is the last frequency returned from UserTimerRequestFrequency. The countdown stops at 0 and the timer is triggered.

Any count in progress is aborted by this function.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimerNumber</td>
<td>Number of a timer previously claimed.</td>
</tr>
<tr>
<td>Count</td>
<td>Number of ticks before a timeout is triggered. The range may vary depending on the hardware implementation of the counter.</td>
</tr>
</tbody>
</table>

Return Value

Possible return values are represented by macros defined in usertimers.h:

- USER_TIMER_OK - Successfully claimed or released timer
- USER_TIMER_INVALID_TIMER - Timer number not present on system
- USER_TIMER_NOTCLAIMED - Another application already has claimed the timer so this application could not claim it
- USER_TIMER_SETFAILED - Unable to set timer
- USER_TIMER_INVALIDPARAMETER - Invalid value

Portability

This function is specific to the RAC6182 hardware.
UserTimerStop

This function stops the user timer. It is prototyped in usertimers.h.

Syntax

```c
#include <Windows.h>
#include <usertimers.h>

DWORD UserTimerStop(DWORD TimerNumber)
```

Remarks

Aborts any currently active countdown in the timer specified by TimerNumber. TimerNumber must refer to a previously claimed timer.

Return Value

Possible return values are represented by macros defined in usertimers.h:

- USER_TIMER_OK - Successfully claimed or released timer
- USER_TIMER_INVALID_TIMER - Timer number not present on system
- USER_TIMER_NOT_CLAIMED - Another application already has claimed the timer so this application could not claim it
- USER_TIMER_NOT_RUNNING - Timer not running

Portability

This function is specific to the RAC6182 hardware

See Also

UserTimerSet

UserTimerGetValue

This function gets the count of user timer. It is prototyped in usertimers.h.

Syntax

```c
#include <Windows.h>
#include <usertimers.h>

DWORD UserTimerGetValue(DWORD TimerNumber, DWORD *Count)
```

Remarks

Requests the current countdown value of an active timer.
Parameter Description
---
TimerNumber Number of a previously claimed timer.
Count Pointer to a DWORD allocated by the caller in which the current count will be stored.

Return Value
Possible return values are represented by macros defined in usertimers.h:

- USER_TIMER_OK - Successfully claimed or released timer
- USER_TIMER_INVALID_TIMER - Timer number not present on system
- USER_TIMER_NOTCLAIMED - Another application already has claimed the timer so this application could not claim it
- USER_TIMER_INVALIDPARAMETER - Invalid value
- USER_TIMER_NOTRUNNING - Timer not running

Portability
This function is specific to the RAC6182 hardware.

UserTimerGetWaitEvent
This function registers to receive notification of timeout event. It is prototyped in usertimers.h.

Syntax
```
#include <usertimers.h>

DWORD UserTimerGetWaitEvent(DWORD TimerNumber, BOOL ManualReset, HANDLE *WaitEvent)
```

Remarks
Creates an event handle in *WaitEvent which may be used in a WaitForSingleObject() or WaitForMultipleObjects() call. This event is set whenever the timer counts down to zero, allowing interrupt driven timer handling.

Parameter Description
---
TimerNumber Number of a previously claimed timer.
ManualReset
WaitEvent Pointer to a caller allocated HANDLE, in which an event handle will be stored.
Return Value

Possible return values are represented by macros defined in usertimers.h:

- USER_TIMER_OK - Successfully claimed or released timer
- USER_TIMER_INVALID_TIMER - Timer number not present on system
- USER_TIMER_NOT_CLAIMED - Another application already has claimed the timer so this application could not claim it
- USER_TIMER_INVALID_PARAMETER - Invalid value

Portability

This function is specific to the RAC6182 hardware.

Functions for Accessing the Hardware Monitor

The RAC6182 platform provides a hardware monitor driver that can be called by applications to monitor the status of various board parameters, to set "warning levels", for these parameters, and to receive warning event signals when values of the monitored parameters fall outside the warning levels. In addition, functions for power-fail monitoring, board reset and reboot are provided here.

The parameters that can be monitored are:

- 12-Volt power supply voltage
- 5-Volt power supply voltage
- 3.3-Volt power supply voltage
- 3-Volt battery voltage
- Board temperature
- Power fail

Note that some of the monitored parameters are of fundamentally different types. Therefore the units of the parameters, will vary according to the parameter being monitored. For example, when an application sets the warning levels for the 5V-power supply monitor, the units of the levels specified in HardwareMonitor API functions will be Volts. However, when an application sets board temperature warning levels, the application must specified temperature in degrees Celsius. Parameter units are described in more detail below for each API function.
The functions exported by the HardwareMonitor driver are listed below.

**hm_GetMonitorLevel**

This function gets the value of monitored parameter. It is prototyped in HardwareMonitorAPI.h.

**Syntax**

```c
#include <Windows.h>
#include <HardwareMonitorAPI.h>

BOOL hm_GetMonitorLevel (DWORD dwMonitorID, double *plfMonitorLevel)
```

**Remarks**

This function queries the hardware monitor for the value of the parameter specified with dwMonitorID. The value is returned as a double precision float in *plfMonitorLevel. The units vary depending on the parameter.

Note that there are no readable values associated with the power fail monitor.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dwMonitorID</td>
<td>ID of monitored parameter whose current value is being queried. Note that only one monitored parameter can be specified. Possible values are represented by macros defined in HardwareMonitorAPI.h as follows: MONITOR_ID_SUPPLY_3V, MONITOR_ID_SUPPLY_5V, MONITOR_ID_SUPPLY_12V, MONITOR_ID_SUPPLY_BATTERY, MONITOR_ID_TEMPERATURE_BOARD</td>
</tr>
<tr>
<td>plfMonitorLevel</td>
<td>Pointer to an application-allocated double precision floating point value which will receive the specified monitor's current level. Power supply values are given in units of voltage. Temperature values are given in units of degrees Celsius.</td>
</tr>
</tbody>
</table>

**Return Value**

Returns TRUE if the query succeeded. FALSE is returned if the query failed.

**Portability**

This function is specific to the RAC6182 hardware.

**See Also**

hm_GetMonitorWarningLevels and hm_SetMonitorWarningLevels
hm_GetMonitorWarningLevels

This function gets warning levels for monitored parameter. It is prototyped in HardwareMonitorAPI.h.

Syntax

```c
#include <Windows.h>
#include <HardwareMonitorAPI.h>

BOOL hm_GetMonitorWarningLevels (DWORD dwMonitorID, double *plfUpperWarningLevel, double *plfLowerWarningLevel)
```

Remarks

This function queries the current warning levels defined for the parameter specified with dwMonitorID. Upper and lower warning levels specify the upper and lower bounds of the monitored parameter during normal operation. If the parameter deviates from the defined operating bounds, it will enter the "warning state".

Power fail is special in that it only has a lower warning level. IfUpperWarningLevel will be the value of the macro MONITOR_WARNING_LEVEL_UNDEFINED.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dwMonitorID</td>
<td>ID of monitored parameter whose warning levels are being set. monitored parameter can be specified. Note that only one monitor ID can be specified. Possible values are represented by macros defined in HardwareMonitorAPI.h as follows:</td>
</tr>
<tr>
<td></td>
<td>MONITOR_ID_SUPPLY_3V</td>
</tr>
<tr>
<td></td>
<td>MONITOR_ID_SUPPLY_5V</td>
</tr>
<tr>
<td></td>
<td>MONITOR_ID_SUPPLY_12V</td>
</tr>
<tr>
<td></td>
<td>MONITOR_ID_SUPPLY_BATTERY</td>
</tr>
<tr>
<td></td>
<td>MONITOR_ID_TEMPERATURE_BOARD</td>
</tr>
<tr>
<td></td>
<td>MONITOR_ID_POWER_FAIL</td>
</tr>
<tr>
<td>plfUpperWarningLevel</td>
<td>Pointer to an application-allocated double-precision floating point where the current upper limit value will be written. The value represents volts or degrees Celsius, depending on the specified parameter. MONITOR_WARNING_LEVEL_UNDEFINED, a macro defined in HardwareMonitorAPI.h, will be written if the upper level bound has not been defined and will not be used to determine if the monitor has entered the warning state.</td>
</tr>
</tbody>
</table>
## Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>plfLowerWarningLevel</td>
<td>Pointer to an application-allocated double-precision floating point where the current lower limit value will be written. The value represents volts or degrees Celsius, depending on the specified parameter. MONITOR_WARNING_LEVEL_UNDEFINED, a macro defined in HardwareMonitorAPI.h, will be written if the upper level bound has not been defined and will not be used to determine if the monitor has entered the warning state.</td>
</tr>
</tbody>
</table>

### Return Value

Returns TRUE if the warning levels were successfully queried. Returns FALSE on failure.

### Portability

This function is specific to the RAC6182 hardware.

### See Also

hm_SetMonitorWarningLevels

### hm_SetMonitorWarningLevels

This function sets warning levels for monitored parameter. It is prototyped in HardwareMonitorAPI.h.

#### Syntax

```c
#include <Windows.h>
#include <HardwareMonitorAPI.h>

BOOL hm_SetMonitorWarningLevels (DWORD dwMonitorID, double lfUpperWarningLevel, double lfLowerWarningLevel)
```

#### Remarks

This function sets upper and lower "warning levels" the parameter specified with dwMonitorID. Upper and lower warning levels specify the upper and lower bounds of the monitored parameter during normal operation. If the parameter deviates from the specified operating bounds, it will enter the "warning state".

The power fail monitor is a 12V supply monitor, but separate from the regular 12V supply monitor. It is special in that it is used not merely to trigger a warning, but also to initiate a system shutdown in case the 12V line drops below the warning level. The power fail monitor has only a lower warning level. If setting levels for power fail, the macro MONITOR_WARNING_LEVEL_UNDEFINED should be used to evaluate lfUpperWarningLevel.
ATTENTION: When power fail lower warning level is set, and the 12V supply subsequently drops below that level, the system will enter a warning state, but it will also begin an irreversible shut down. Care should be taken not to set regular 12-Volt low warning level lower than the power fail level; otherwise, it will not be possible to detect a warning in case the 12V drops below its lower level. Also, care must be taken not to set the power fail level higher than the current 12-Volt level; otherwise, the system will immediately enter the power fail shutdown state.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dwMonitorID</td>
<td>ID of monitored parameter whose limits are being set. Note that only one monitored parameter can be specified. Possible values are represented by macros defined in HardwareMonitorAPI.h as follows: MONITOR_ID_SUPPLY_3V MONITOR_ID_SUPPLY_5V MONITOR_ID_SUPPLY_12V MONITOR_ID_SUPPLY_BATTERY MONITOR_ID_TEMPERATURE_BOARD MONITOR_ID_POWER_FAIL</td>
</tr>
<tr>
<td>lfUpperWarningLevel</td>
<td>Double precision floating point level defining the upper bound of the parameter during normal operation. The given value represents voltage or degrees Celsius, depending on the specified parameter. If MONITOR_WARNING_LEVEL_UNDEFINED, a macro defined in HardwareMonitorAPI.h, is specified, the upper level bound will be undefined and not used to determine if the monitor enters the warning state.</td>
</tr>
<tr>
<td>lfLowerWarningLevel</td>
<td>Double precision floating point level defining the lower bound of the monitor parameter during normal operation. The given value represents volts or degrees Celsius, depending on the specified parameter. If MONITOR_WARNING_LEVEL_UNDEFINED, a macro defined in HardwareMonitorAPI.h, is specified, the lower level bound will be undefined and not used to determine if the monitor enters the warning state.</td>
</tr>
</tbody>
</table>

Return Value

Returns TRUE if the warning levels were successfully set. Returns FALSE on failure.
Portability
This function is specific to the RAC6182 hardware.

See Also
hm_GetMonitorWarningLevels

hm_RegisterMonitorWarningEvent
This function registers to receive a warning of a parameter out-of-limit. It is prototyped in HardwareMonitorAPI.h.

Syntax
#include <Windows.h>
#include <HardwareMonitorAPI.h>

BOOL hm_RegisterMonitorWarningEvent (DWORD dwMonitorIDMask, HANDLE *phEventHandle)

Remarks
An application that needs to be notified when one or more monitor parameters enter the warning state should register for an event with this function. The caller specifies, via dwMonitorIDMask, what parameters should be used to trigger a warning event when their values exceed warning levels. An event handle is passed to the caller in *phEventHandle.

Note: It is possible to set an event for power fail (to trigger some clean up before system shutdown), but this event must be a separate event from that used for general warnings.

If this function succeeds, the caller can wait for the event using one of the standard Win32 WaitForxxx() functions. Once the event is triggered and the caller’s thread falls through the WaitForxxx() function, the caller can determine which monitor sources are currently in the warning state via the function hm_GetMonitorWarnings. If any of the monitor sources in question are still in a warning state, the caller can act accordingly.

Note that monitor parameters will vary with time, and may oscillate about a defined upper or lower warning level for a short period. Therefore, when a warning state event has been triggered, the calling application should poll the warning status of any monitor sources of concern (using hm_GetMonitorWarnings) for a while to ensure that the monitor source remains in a warning state before acting. Note also that, to avoid oscillating events due to a lack of input hysteresis, the hardware monitor driver will not signal an event when a monitor source leaves the warning state. Applications must poll the device's warning state to determine when/if it resumes normal operation.
The phEventHandle returned by this function is a standard Win32 auto-reset event handle. However, an application should NOT close the handle using the Win32 CloseHandle function. Instead the application should close the handle and unregister the event using the hm_UnregisterMonitorWarningEvent function.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dwMonitorIDMask</td>
<td>Bitmask combination of all of the monitor sources that will trigger the returned event when entering the warning state.</td>
</tr>
<tr>
<td>MONITOR_ID_SUPPLY_3V</td>
<td></td>
</tr>
<tr>
<td>MONITOR_ID_SUPPLY_5V</td>
<td></td>
</tr>
<tr>
<td>MONITOR_ID_SUPPLY_12V</td>
<td></td>
</tr>
<tr>
<td>MONITOR_ID_SUPPLY_BATTERY</td>
<td></td>
</tr>
<tr>
<td>MONITOR_ID_TEMPERATURE_BOARD</td>
<td></td>
</tr>
<tr>
<td>MONITOR_ID_POWER_FAIL</td>
<td></td>
</tr>
<tr>
<td>phEventHandle</td>
<td>Pointer to an application-allocated HANDLE.</td>
</tr>
</tbody>
</table>

**Return Value**

Returns TRUE if the monitor warning event has been successfully registered. Returns FALSE on failure.

**Portability**

This function is specific to the RAC6182 hardware.

**See Also**

hm_UnregisterMonitorWarningEvent

**hm_UnregisterMonitorWarningEvent**

This function cancels the registration for a warning of a parameter out-of-limit. It is prototyped in HardwareMonitorAPI.h.

**Syntax**

```c
#include <Windows.h>
#include <HardwareMonitorAPI.h>

BOOL hm_UnregisterMonitorWarningEvent (HANDLE hEventHandle)
```
Remarks
This function unregisters and frees a warning state notification event that had previously been created using hm_RegisterMonitorWarningEvent. This function will automatically free hEventHandle, so the application should not attempt to free it with CloseHandle.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hEventHandle</td>
<td>Handle of the previously registered warning event that is now being unregistered and freed.</td>
</tr>
</tbody>
</table>

Return Value
Returns TRUE if the monitor warning event has been successfully unregistered. Returns FALSE on failure.

Portability
This function is specific to the RAC6182 hardware.

See Also
hm_RegisterMonitorWarningEvent

hm_GetMonitorWarnings
This function gets warnings. It is prototyped in HardwareMonitorAPI.h.

Syntax
#include <Windows.h>
#include <HardwareMonitorAPI.h>

BOOL hm_GetMonitorWarnings (DWORD *pdwMonitorID)

Remarks
This function returns a bitwise OR’d combination of all monitor sources that are currently in the warning state. Note that this function does not latch any previous warning states that may have previously triggered warning notification events. Therefore, if a monitor source enters a warning state and triggers a notification event, it is possible that the monitor has left the warning state before an application calls this function.

Note: A system shutdown on account of a power failure cannot be sensed via this call. When the system is in the power fail shutdown state, communication with the microcontroller that this function uses to get monitor levels is already shutdown. To sense power fail an application must use hm_RegisterMonitorWarningEvent to register an event with the power fail monitor.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pdwMonitorID</td>
<td>Pointer to an application-allocated DWORD that will receive a bitmask combination of all monitor sources currently in the warning state. The following macros defined in HardwareMonitorAPI.h can be used to test for specific parameters: MONITOR_ID_SUPPLY_3V MONITOR_ID_SUPPLY_5V MONITOR_ID_SUPPLY_12V MONITOR_ID_SUPPLY_BATTERY MONITOR_ID_TEMPERATURE_BOARD</td>
</tr>
</tbody>
</table>

Return Value

Returns TRUE if the function succeeds. Returns FALSE on failure.

Portability

This function is specific to the RAC6182 hardware.

hm_RebootBoard

This function reboots the system. It is prototyped in HardwareMonitorAPI.h.

Syntax

#include <Windows.h>
#include <HardwareMonitorAPI.h>

BOOL hm_RebootBoard (void)

Remarks

This function performs a reboot the RAC6182 board. The reboot reason code is set, caches are flushed, and a full reboot is performed, including reset of hardware chips. Applications should use this function instead of alternatives such as using KernelIoControl to reset the board.

Return Value

Returns FALSE on failure. Function will not return on success as the board will reset.

Portability

This function is specific to the RAC6182 hardware.
hm_GetBootReason

This function gets the reason for last boot. It is prototyped in HardwareMonitorAPI.h.

Syntax
#include <Windows.h>
#include <HardwareMonitorAPI.h>

DWORD hm_GetBootReason (void)

Remarks
This function returns a DWORD representing the reason the board was last booted. Magic cookies and intelligent selection of default cases are used to distinguish as many scenarios as possible. However, the method relies on the probability that DRAM will hold a value for 10s of milliseconds without refresh, and on other such factors. Accordingly, the returned value cannot be guaranteed to be correct, especially in cases of very brief power interruptions.

Return Value
Return values are represented by macros defined in HardwareMonitorAPI.h.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOT_REASON_UNTRAPPED</td>
<td>Board was reset after a power-up boot, not by a user or operating system call, but most likely by the watchdog.</td>
</tr>
<tr>
<td>BOOT_REASON_COLD_POWER_CYCLE</td>
<td>Board went through a normal power-up boot.</td>
</tr>
<tr>
<td>BOOT_REASON_POWER_DOWN_CLEAN</td>
<td>Board was previously powered down when power fail monitor level was set, and either no applications were registered for power down event handling, or all applications registered completed their shutdown activities and signaled this to the operating system.</td>
</tr>
<tr>
<td>BOOT_REASON_POWER_DOWN_DIRTY</td>
<td>Board was previously power down when power fail monitor level was set, but at least one application registered for a power down event did not signal that it completed its shutdown activities.</td>
</tr>
<tr>
<td>BOOT_REASON_WARM_UNDEFINED</td>
<td>Board was reset for an unknown reason</td>
</tr>
<tr>
<td>BOOT_REASON_WARM_REQUESTED</td>
<td>Board was reset by an application call to hm_RebootBoard.</td>
</tr>
<tr>
<td>BOOT_REASON_WARM_INTERNAL</td>
<td>Board was reset by some operating system operation</td>
</tr>
</tbody>
</table>

Portability
This function is specific to the RAC6182 hardware.
The RAC6182 software includes a Retentive Memory driver that can be called by applications to read and write to the RAC6182’s battery-backed RAM. The driver is implemented as a dynamic link library ().

Retentive Memory driver functions available to applications include functions to lock and unlock the battery-backed RAM. When an application holds a lock on this RAM, no other application can read or write data. There are also functions that allow applications to verify memory contents using a checksum.

The functions exported by the RetentiveMem driver are listed below.

### RetMemLock

This function locks retentive memory. It is prototyped in RetentiveMemAPI.h.

#### Syntax

```c
#include <Windows.h>
#include <RetentiveMemAPI.h>

BOOL RetMemLock(DWORD dwTimeout)
```

#### Remarks

This function locks the entire retentive memory (battery-backed RAM), preventing other applications from reading and writing. An application must have the lock on retentive memory before any other retentive memory function will complete successfully; thus, this function should always be called, and its return value checked, before any other function retentive memory function is called.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DwTimeout</td>
<td>Number of milliseconds to wait for lock before failing.</td>
</tr>
</tbody>
</table>

#### Return Value

Returns TRUE if the lock request succeeded. Returns FALSE if the lock request failed because another application did not release the lock within dwTimeout milliseconds.

#### Portability

This function is specific to the RAC6182 hardware.

#### See Also

RetMemUnlock
**RetMemUnlock**

This function unlocks retentive memory. It is prototyped in RetentiveMemAPI.h.

**Syntax**

```c
#include <Windows.h>
#include <RetentiveMemAPI.h>

BOOL RetMemUnlock (void)
```

**Remarks**

The application should always call this function to release a lock on retentive memory before exiting.

**Return Value**

Returns TRUE if the lock is released. Returns FALSE on failure.

**Portability**

This function is specific to the RAC6182 hardware.

**See Also**

RetMemLock

---

**RetMemWrite**

This function writes to retentive memory. It is prototyped in RetentiveMemAPI.h.

**Syntax**

```c
#include <Windows.h>
#include <RetentiveMemAPI.h>

DWORD RetMemWrite (DWORD dwOffset, DWORD dwLength, BYTE *pbtBuffer)
```

**Remarks**

This function copies dwLength bytes of data from a user data buffer to a specific area in retentive memory. If less than dwLength bytes are available to be written, the number of available bytes are copied. In this case, the number of bytes returned will be less than dwLength. If another process has the memory locked, no bytes will be written and the return value will be 0.

The total amount of available memory is represented by the macro RET_MEM_MEMORY_SIZE, defined in RetentiveMemAPI.h.
Parameter Description
---
dwOffset | Starting offset in RAM at which data will be written. Offset 0 is the first byte of RAM.
dwLength | Number of bytes to be written.
pbtBuffer | Pointer to a buffer containing data to be written to the RAM.

**Return Value**
Returns the total number of bytes actually written

**Portability**
This function is specific to the RAC6182 hardware.

**See Also**
RetMemRead

### RetMemRead

This function reads retentive memory. It is prototyped in RetentiveMemAPI.h.

**Syntax**
```
#include <Windows.h>
#include <RetentiveMemAPI.h>
DWORD RetMemRead (DWORD dwOffset, DWORD dwLength, BYTE *pbtBuffer)
```

**Remarks**
This function copies dwLength bytes of data from an area of retentive memory into a user data buffer. The number of bytes actually read is returned. If fewer than dwLength bytes are available, the value returned will be less than dwLength. If another process has the memory locked, no bytes will be read and the return value will be 0.

The total amount of available memory (in bytes) is given by the macro RET_MEM_MEMORY_SIZE, defined in RetentiveMemAPI.h.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dwOffset</td>
<td>Starting offset in RAM of data to be read. Offset 0 is the first byte of RAM.</td>
</tr>
<tr>
<td>dwLength</td>
<td>Number of bytes to be read.</td>
</tr>
<tr>
<td>pbtBuffer</td>
<td>Pointer to the beginning of a buffer to be filled with data from RAM.</td>
</tr>
</tbody>
</table>
Return Value
Returns the number of bytes actually read.

Portability
This function is specific to the RAC6182 hardware.

See Also
RetMemWrite

RetMemVerifyMemory
This function verifies retentive memory. It is prototyped in RetentiveMemAPI.h.

Syntax
#include <Windows.h>
#include <RetentiveMemAPI.h>
BOOL RetMemVerifyMemory (void)

Remarks
This function calculates the checksum of retentive memory and compares it to the master checksum. If the two values do not match, the function returns FALSE, indicating the existence of corrupt data in the memory. If the two values match, the function returns TRUE, indicating that data are valid.

This function does not rewrite the master checksum; therefore, successive calls to this function will return the same result.

Applications should call this function before reading from or writing to the battery-backed RAM, to verify that its data are valid. The caller must hold a lock on the RAM in order to use this function.

Return Value
Returns TRUE if the calculated checksum matches the master checksum. Returns FALSE if the checksums do not match or if another application has locked memory.

Portability
This function is specific to the RAC6182 hardware.

See Also
RetMemCalculateChecksum
RetMemCalculateChecksum

This function calculates retentive memory checksum. It is prototyped in RetentiveMemAPI.h.

Syntax

```c
#include <Windows.h>
#include <RetentiveMemAPI.h>

BOOL RetMemCalculateChecksum (void)
```

Remarks

This function recalculates the checksum of retentive memory and writes it as the master checksum.

![ATTENTION:]

ATTENTION: If any data in the retentive memory are corrupt, as indicated by a return of FALSE from a call to RetMemVerifyMemory, the corruptions should be repaired before calling this function. Any data in the memory that are corrupt at the time this function is called will remain corrupt, and the recalculated master checksum based on the corrupt data will render the corruptions invisible to subsequent calls of RetMemVerifyMemory.

If the caller cannot repair corruptions, because they have to do with data maintained by other applications, the caller should at least ensure that all other applications are informed that their data may not be valid prior to issuing this call. Otherwise, some applications might interpret bad data as if it were good.

Return Value

Returns TRUE if the checksum is recalculated. Returns FALSE if another process has locked memory.

Portability

This function is specific to the RAC6182 hardware.

See Also

RetMemVerifyMemory
Applications can utilize the RAC6182 platform's serial ports via the standard WIN32 API's "File I/O" interface. A complete description of the File I/O interface is outside the scope of this document. Users are referred to the Microsoft Win32 SDK documentation for additional information. However, some of the arguments needed by the File I/O functions for serial port control are specific to the RAC6182. These arguments are represented by macros defined in the c header file othersdk.h, and are treated here in detail.

### DeviceIoControl

This function accesses the serial port. It is prototyped in Winbase.h (included in Windows.h).

#### Syntax

```c
#include <Windows.h>
#include <othersdk.h>

BOOL DeviceIoControl(
    HANDLE hDevice,
    DWORD dwIOControlCode,
    LPVOID lpInBuffer,
    DWORD nInBufferSize,
    LPVOID lpOutBuffer,
    DWORD nOutBufferSize,
    LPDWORD lpBytesReturned,
    LPOVERLAPPED lpOverlapped
)
```

#### Remarks

See the documentation in the Microsoft Windows CE development kit for additional information.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDevice</td>
<td>Handle to device (returned by CreateFile)</td>
</tr>
<tr>
<td>dwIOControlCode</td>
<td>Evaluate with one of the macros defined in othersdk.h. (See macro descriptions that follow.)</td>
</tr>
<tr>
<td>lpInBuffer</td>
<td>Pointer to input data buffer. Used for set operations</td>
</tr>
<tr>
<td>nInBufferSize</td>
<td>Size of input data buffer. Used for set operations.</td>
</tr>
<tr>
<td>lpOutBuffer</td>
<td>Pointer to output data buffer. Used for get operations.</td>
</tr>
<tr>
<td>nOutBufferSize</td>
<td>Size of output data buffer. Used for get operations.</td>
</tr>
<tr>
<td>lpBytesReturned</td>
<td>Pointer to count of bytes returned</td>
</tr>
<tr>
<td>lpOverlapped</td>
<td>Pointer to overlapped information (NULL for CE)</td>
</tr>
</tbody>
</table>
### Macro Description

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOCTL_SERIAL_SET_PORT_MODE</td>
<td>Sets the electrical interface for the specified serial port. Note that only COM1 supports switching electrical interfaces. lpInBuffer should point to a caller allocated WORD containing the code for the desired electrical interface. The possible codes are represented by the following macros defined in othersdk.h: SERIAL_MODE_RS232 SERIAL_MODE_RS485 SERIAL_MODE_RS422 nInBufferSize should be sizeof(WORD).</td>
</tr>
<tr>
<td>IOCTL_SERIAL_QUERY_PORT_MODE</td>
<td>Returns the current electrical interface setup for the specified serial port. lpOutBuffer should point to a caller allocated WORD where the code for the current mode will be stored. The value of *lpOutBuffer can be decoded by comparison with one of the following macros: SERIAL_MODE_RS232 SERIAL_MODE_RS485 SERIAL_MODE_RS422 nOutBufferSize should be sizeof(WORD).</td>
</tr>
<tr>
<td>IOCTL_SERIAL_ENABLE_TX_FIFO</td>
<td>Enable the transmit FIFO buffer for the specified serial port. Note that this is a 16 byte buffer. lpInBuffer should point to a caller allocated BOOL, which should contain TRUE or FALSE as required to enable or disable the FIFO. nInBufferSize should be sizeof(BOOL).</td>
</tr>
<tr>
<td>IOCTL_SERIAL_ENABLE_RX_FIFO</td>
<td>Enable the receiver FIFO buffer for the specified serial port. Note that this is a 16 byte buffer. lpInBuffer should point to a caller allocated BOOL, which should contain TRUE or FALSE as required to enable or disable the FIFO. nInBufferSize should be sizeof(BOOL).</td>
</tr>
<tr>
<td>IOCTL_SERIAL_SET_TX_FIFO_WATERMARK</td>
<td>Do not use. Not supported on RAC6182.</td>
</tr>
<tr>
<td>IOCTL_SERIAL_SET_RX_FIFO_WATERMARK</td>
<td>Do not use. Not supported on RAC6182.</td>
</tr>
<tr>
<td>IOCTL_SERIAL_QUERY_TX_FIFO_COUNT</td>
<td>Do not use. Not supported on RAC6182.</td>
</tr>
<tr>
<td>IOCTL_SERIAL_QUERY_RX_FIFO_COUNT</td>
<td>Do not use. Not supported on RAC6182.</td>
</tr>
<tr>
<td>IOCTL_SERIAL_SET_COUNTRY_CODE</td>
<td>Do not use. Not supported on RAC6182.</td>
</tr>
<tr>
<td>IOCTL_SERIAL_QUERY_COUNTRY_CODE</td>
<td>Do not use. Not supported on RAC6182.</td>
</tr>
</tbody>
</table>
Application Interface to Output Debug Messages

The RAC6182 provides support for an external debug monitor connected at the second serial port (COM2). Normally, the second serial port is configured as a standard communications port; in this case, it does not handle system debug messages, which, if generated, are discarded. However, this port can be specially set up to pass system debug messages to an external device; in this case, the second serial port becomes unavailable to applications for normal communications use. Details for the setup of the port are given in an appendix to this manual.

A debug monitor can be very useful for Windows CE operating system developers, since it can capture debug messages emitted by the operating system by way of a serial port, even before all Windows CE services, including display services, are operational.

Application developers can also make use of these facilities, provided their programs written to produce debug output via the NKDbgPrintfW() function described below. However, since the use of an external debug monitor requires a special configuration of the RAC6182, it is recommended that the use of external debug output be limited to logging fatal or unusual error conditions. If an application developer believes a fatal error is occurring, the RAC6182 can be set up to pass error messages that might allow him or her to diagnose the problem more easily.

When debug mode is enabled, applications should not open COM2 or they will interfere with debug output messages. When debug mode is not enabled, all debug output is silently ignored and COM2 is available for general application use.
**NKDbgPrintfW**

This function generates output to an external debug monitor. It is prototyped in dbgapi.h.

**Syntax**

```c
#include <Windows.h>
#include <dbgapi.h>

void NKDbgPrintfW(LPWSTR lpszFmt, ...)
```

**Remarks**

This is a var-args function similar to any of the printf() family, which takes a Unicode formatting string and zero or more other arguments. dbgapi.h defines macros RETAILMSG() and DEBUGMSG() which may be useful in setting up conditional debug outputs.

**Return Value**

Nothing

**Portability**

This function is specific to the RAC6182 hardware.
## Appendix A

### Operating System Files

**Table J**

<table>
<thead>
<tr>
<th>System Executables in Windows</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>async.asy (ActiveSync)</td>
<td></td>
</tr>
<tr>
<td>cmd.exe</td>
<td></td>
</tr>
<tr>
<td>control.exe</td>
<td></td>
</tr>
<tr>
<td>ctlpnl.exe</td>
<td>Control Panel</td>
</tr>
<tr>
<td>explorer.exe (Windows Explorer)</td>
<td>Windows Explorer</td>
</tr>
<tr>
<td>flashavr.exe</td>
<td></td>
</tr>
<tr>
<td>format.exe</td>
<td></td>
</tr>
<tr>
<td>iesample.exe (Internet Explorer)</td>
<td></td>
</tr>
<tr>
<td>LocalOSUpdate.exe</td>
<td></td>
</tr>
<tr>
<td>osmonitor.exe</td>
<td></td>
</tr>
<tr>
<td>pegterm.exe (Terminal)</td>
<td></td>
</tr>
<tr>
<td>ping.exe</td>
<td></td>
</tr>
<tr>
<td>rapisvr.exe</td>
<td></td>
</tr>
<tr>
<td>regflush.exe</td>
<td>Saves RAM based registry to persistent storage</td>
</tr>
<tr>
<td>remnet.exe (Remote Networking)</td>
<td></td>
</tr>
<tr>
<td>repllog.exe (PC Link)</td>
<td>Used to establish communications with a host f</td>
</tr>
<tr>
<td>rnaapp.exe</td>
<td></td>
</tr>
<tr>
<td>wplayer.exe</td>
<td></td>
</tr>
<tr>
<td>system dynamic link libraries in Windows</td>
<td></td>
</tr>
<tr>
<td>IECEExt.dll</td>
<td></td>
</tr>
<tr>
<td>mlang.dll</td>
<td></td>
</tr>
<tr>
<td>mscefile.dll</td>
<td></td>
</tr>
<tr>
<td>mshtml.dll</td>
<td></td>
</tr>
<tr>
<td>rsabase.dll</td>
<td></td>
</tr>
<tr>
<td>shdocvw.dll</td>
<td></td>
</tr>
<tr>
<td>shlwapi.dll</td>
<td></td>
</tr>
<tr>
<td>urlmon.dll</td>
<td></td>
</tr>
<tr>
<td>wininet.dll</td>
<td></td>
</tr>
</tbody>
</table>
## Memory Usage

### Table K
**RAM Usage**

<table>
<thead>
<tr>
<th>Component</th>
<th>Memory Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM FS space - Core OS components (kernel, networking, drivers)</td>
<td>4.0 MB</td>
</tr>
<tr>
<td>RAM FS space - Windows desktop (commctrl, explorer, ctl panel, fonts)</td>
<td>3.0 MB</td>
</tr>
<tr>
<td>RAM FS space – Internet explorer (browser object, HTML, javascript)</td>
<td>5.0 MB</td>
</tr>
<tr>
<td>Reserved/global variable area used by drivers, kernel</td>
<td>64 KB</td>
</tr>
<tr>
<td>Network DMA buffers</td>
<td>128 KB</td>
</tr>
<tr>
<td>USB DMA buffers, private memory</td>
<td>32 KB</td>
</tr>
<tr>
<td>Heap and stack used by core OS components</td>
<td>1.0 MB</td>
</tr>
<tr>
<td>Heap and stack used by Windows desktop components</td>
<td>1.0 MB</td>
</tr>
<tr>
<td>Heap and stack used by Internet Explorer</td>
<td>varies greatly</td>
</tr>
</tbody>
</table>

### Table L
**Flash (Disk on Chip) Usage**

<table>
<thead>
<tr>
<th>Component</th>
<th>Memory Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core OS + Windows desktop= 7.0M* 2/3 for LZ compression</td>
<td>4.7MB</td>
</tr>
<tr>
<td>Core OS + desktop+IE=12.0M*2/3 for LZ compression</td>
<td>8.0MB</td>
</tr>
<tr>
<td>Space for compressed non-volatile registry (2 1.0M areas for safety)</td>
<td>2.0MB</td>
</tr>
</tbody>
</table>
Connecting an External Debug Monitor

The RAC6182 provides support for an external debug monitor connected at the second serial port.

A debug monitor can be very useful for Windows CE operating system developers, since it can capture debug messages emitted by the operating system by way of a serial port, even before all Windows CE services, including display services, are operational.

Application developers can also make use of these facilities, provided their programs written to produce debug output via the NKDbgPrintfW() function described elsewhere in this manual.

In order to cause the operating system to direct debug information emitted via this function to the second serial port, it is necessary to perform either one of the following setups:

- Set the debug jumper associated with the boot ROM on the system circuit board
- Set the system debug flag (by means of a program that calls the system function rm_SetParameter)

Once the system has been set up to use COM2 as the debug port, it is necessary to connect an external debug monitor to this port. Any display device with a serial interface can be used as a monitor. However, the most common choice will be a desktop computer running Windows and a Windows communications application such as Hyperterminal.

The RAC6182 should be connected to the debug monitor using a null modem cable. (But note that it might be necessary to install a jumper at the serial port connector that attaches to the RAC6182.)

The monitor device should be set up for serial communications as follows:

- 57,600 Baud
- 8 Data Bits
- No Parity
- 1 Stop Bit
- Hardware Flow Control
Index

B
Bezel ID EEPROM, 1-18

C
Central processing unit (CPU), 1-1

D
Debug messages, 4-60

F
Files
C/C++ Development Kit, 3-2
local file systems, 1-11
Windows CE Registry, 1-9
Functions
bezel EEPROM
be_GetBezelEEPROMParameter, 4-7
be_ReadBezelEEPROMParameter, 4-10
be_SetBezelEEPROMParameter, 4-8
be_WriteBezelEEPROMParameter, 4-11
custom keypad handlers
KhDeinitialize, 4-13
KhGetGlobalKeySettings, 4-13
KhGetKeyAttributeTable, 4-13
KhInitialize, 4-13
KhSetGlobalKeySettings, 4-13
KhTranslateVKey, 4-14
debug messages
NKDbgPrintW, 4-61
DRAM
SetSystemMemoryDivision, 4-30
hardware monitors
hm_GetMonitorLevel, 4-43
hm_GetMonitorWarningLevels, 4-44
hm_GetMonitorWarnings, 4-50
hm_GetRebootReason, 4-52
hm_RebootBoard, 4-51
hm_RegisterMonitorWarningEvent, 4-48
hm_SetMonitorWarningLevels, 4-45
hm_UnregisterMonitorWarningEvent, 4-49
LED
NledGetDeviceInfo, 4-16
NledSetDevice, 4-18
main
do_ClearBits, 4-5
do_ReadPort, 4-1
do_SetBits, 4-4
do_ToggleBits, 4-6
do_WritePort, 4-2
operating system
osu_RemoteUpdateOSFromRAM, 4-28
osu_UpdateOSFromFile, 4-28
parameters
GetParameter, 4-32
SetParameter, 4-33
PCI drivers
HalGetBusDataByOffset, 4-22
HalSetBusDataByOffset, 4-23
HalTranslateBusAddress, 4-21
InterruptDisable, 4-25
InterruptDone, 4-25
InterruptInitialize, 4-24
MmMaploSpace, 4-24
registry
FlushRegistry, 4-29
retentive memory
RetMemCalculateChecksum, 4-57
RetMemLock, 4-53
RetMemRead, 4-55
RetMemUnlock, 4-54
RetMemVerifyMemory, 4-56
RetMemWrite, 4-54
serial ports
DeviceIoControl, 4-58
timers
Index

UserTimerClaim, 4-35
UserTimerGetNumberOfTimers, 4-35
UserTimerGetValue, 4-40
UserTimerGetWaitEvent, 4-41
UserTimerRequestFrequency, 4-36
UserTimerSet, 4-39
UserTimerStop, 4-40

watchdog timer
Watchdog_Tag, 4-11

H

Hardware
architecture, 1-1
CPU, 1-1
memory devices
battery backed RAM, 1-3
boot ROM, 1-2
Disk-on-Chip, 1-2
DRAM, 1-2
PCI subsystem, 1-5
super I/O, 1-3
host system
setting up
Basic development, 2-5
C/C++ development, 2-3

I

Input device handlers, 1-13
Installation
development system, 2-3
distribution, 2-1
methods, 2-2
persistence considerations, 2-2
RAC6182 SDK, 2-6
setting up host machine
Basic development, 2-5
C/C++ development, 2-3
upgrades, 2-2

M

Memory devices
battery backed RAM, 1-3
boot ROM, 1-2
Disk-on-Chip, 1-2
DRAM, 1-2

O

Operating systems
boot sequence, 1-6
local file systems, 1-11
Windows CE, 1-6

P

PCI bus, 1-20
PCI device
sample code, 4-26

R

RAC6182
hardware
architecture, 1-1
battery backed RAM, 1-3
boot ROM, 1-2
CPU, 1-1
Disk-on-Chip, 1-2
DRAM, 1-2
PCI subsystem, 1-5
super I/O, 1-3
SDK
configuration, 2-8
debug messages, 4-60
files, 3-2
functions, 4-7, 4-11, 4-12, 4-15, 4-19, 4-28, 4-34, 4-42, 4-53
functions for digital output control, 4-1
installing, 2-6
overview, 3-1
streams interface, 4-15, 4-58
software
architecture, 1-6
bezel ID EEPROM, 1-18
boot sequence, 1-6
development system, 2-3
distribution, 2-1
input device handlers, 1-13
installation, 2-2
local file systems, 1-11
operating system, 1-6
PCI bus, 1-20
persistence considerations, 2-2
registry, 1-9
runtime environment, 1-21
system shutdown, 1-22
upgrades, 2-2
Registry, 1-9
Runtime environment, 1-21

S
SDK
configuration, 2-8
debug messages, 4-60
files, 3-2
functions
be_GetBezelEEPROMParameter, 4-7
be_ReadBezelEEPROMParameter, 4-10
be_SetBezelEEPROMParameter, 4-8
be_WriteBezelEEPROMParameter, 4-11
do_ClearBits, 4-5
do_ReadPort, 4-1
do_SetBits, 4-4
do_ToggleBits, 4-6
do_WritePort, 4-2
FlushRegistry, 4-29
GetParameter, 4-32
HalGetBusDataByOffset, 4-22
HalSetBusDataByOffset, 4-23
HalTranslateBusAddress, 4-21
hm_GetMonitorLevel, 4-43
hm_GetMonitorWarningLevels, 4-44
hm_GetMonitorWarnings, 4-50
hm_GetRebootReason, 4-52
hm_RebootBoard, 4-51
hm_RegisterMonitorWarningEvent, 4-48
hm_SetMonitorWarningLevels, 4-45
hm_UnregisterMonitorWarningEvent, 4-49
InterruptDisable, 4-25
InterruptDone, 4-25
InterruptInitialize, 4-24
KhDeinitialize, 4-13
KhGetGlobalKeySettings, 4-13
KhGetKeyAttributeTable, 4-13
KhInitialize, 4-13
KhSetGlobalKeySettings, 4-13
KhTranslateVKey, 4-14
MmMaploSpace, 4-24
NKDbgPrintW, 4-61
NledGetDeviceInfo, 4-16
NledSetDevice, 4-18
osu_RemoteUpdateOSFromRAM, 4-28
osu_UpdateOSFromFile, 4-28
RetMemCalculateChecksum, 4-57
RetMemLock, 4-53
RetMemRead, 4-55
RetMemUnlock, 4-54
RetMemVerifyMemory, 4-56
RetMemWrite, 4-54
SetParameter, 4-33
SetSystemMemoryDivision, 4-30
UserTimerClaim, 4-35
UserTimerGetNumberOfTimers, 4-35
UserTimerGetValue, 4-40
UserTimerGetWaitEvent, 4-41
UserTimerRequestFrequency, 4-36
UserTimerSet, 4-39
UserTimerStop, 4-40
Watchdog_Tag, 4-11
installing, 2-6
overview, 3-1
streams interface
DeviceIoControl, 4-58
Software
  bezel ID EEPROM, 1-18
  boot sequence, 1-6
  input device handlers, 1-13
  local file systems, 1-11
  PCI bus, 1-20
  runtime environment, 1-21
  system shutdown, 1-22
  Windows CE OS, 1-6
  Windows CE Registry, 1-9
IBM is a registered trademark of International Business Machines Corporation.

VGA is a trademark of International Business Machines Corporation.

PC AT is a trademark of International Business Machines Corporation.

Microsoft is a registered trademark of Microsoft Corporation.

Microsoft Windows is a trademark of Microsoft Corporation.

Rockwell Automation helps its customers receive a superior return on their investment by bringing together leading brands in industrial automation, creating a broad spectrum of easy-to-integrate products. These are supported by local technical resources available worldwide, a global network of system solutions providers, and the advanced technology resources of Rockwell.

Worldwide representation.