Reference Manual

P/N 070214-001

2D JANUS™ PSK for C/C++

Intermec

A UNOVA Company

Intermec® Corporation
6001 36th Avenue West
P.O. Box 4280
Everett, WA 98203-9280

U.S. technical and service support: 1-800-755-5505
U.S. media supplies ordering information: 1-800-227-9947

Canadian technical and service support: 1-800-688-7043
Canadian media supplies ordering information: 1-800-268-6936

Outside U.S. and Canada: Contact your local Intermec service supplier.

Contents

3 Software Interrupts

4 Advanced Programming

A Status Codes

B Sample Interrupt Programs

I Index

Before You Begin

This section introduces you to standard warranty provisions, safety precautions, warnings and cautions, document formatting conventions, and sources of additional product information.

Warranty Information

To receive a copy of the standard warranty provision for this product, contact your local Intermec sales organization. In the U.S. call (800) 755-5505, and in Canada call (800) 688-7043. Otherwise, refer to the Worldwide Sales & Service list shipped with this manual for the address and telephone number of your Intermec sales organization.

Cautions

The cautions in this manual use the following format.

Caution
A caution alerts you to an operating procedure, practice, condition, or statement that must be strictly observed to prevent equipment damage or destruction, or corruption or loss of data.

Conseil
Une précaution vous alerte d'une procédure de fonctionnement, d'une méthode, d'un état ou d'un rapport qui doit être strictement respecté pour empêcher l'endommagement ou la destruction de l'équipement, ou l'altération ou la perte de données.

About This Manual

This manual is part of the JANUS Programmer's Software Kit manual set. It describes the special features and methods needed for programming the JANUS family of PC-compatible readers. If you plan to write programs in C or C++, the information in this manual is very valuable.

If you plan to write programs using Interactive Reader Language (IRL), refer to the *IRL Programming Reference Manual*, Part No. 048609, and your JANUS user's manual for programming guidelines.

Organization

The *2D JANUS PSK for C/C++ Reference Manual* is divided into five chapters and two appendixes as described below:

Chapter	What You Will Find
1	*Getting Started* This chapter describes the capabilities of and programming methods for the JANUS readers. It also explains how to install the Programmer's Software Kit Language Libraries disk that is provided with this manual.
2	*C Language Support* This chapter describes how to write C applications using the Programmer's Software Kit (PSK) library functions.
3	*Software Interrupts* This chapter explains how to use software interrupts in your programs.
4	*Advanced Programming* This chapter explains the various reader input modes and how to use the power management software functions.
5	*Using the PSK With DCM* This chapter explains how to program the reader to access a remote database connected through Intermec DCM software.

Chapter	What You Will Find
Appendix A	*Status Codes* This appendix lists status codes that are returned by the PSK functions.
Appendix B	*Sample Interrupt Programs* This appendix lists C language programs that demonstrate how to use software interrupts.

Terms and Conventions

- A *reader* is the JANUS 2010, 2020, or 2050 PC-compatible bar code reader.
- *Reader services* are the functional abilities that distinguish the JANUS readers from a normal PC. For example, the reader's ability to decode bar code data as if it came from a PC keyboard is a typical reader service.
- *Software interrupts* are the synchronous triggering of interrupts used for application program interfaces.
- *Library functions* are the Intermec-specific functions provided in the language libraries you use to invoke various reader services.
- *PSK* means the Programmer's Software Kit and refers to the language libraries and the associated manuals.
- The *Programmer's Software Kit Language Libraries* is the disk shipped with this manual. It contains sample programs and library functions for interfacing with the reader.
- The *keypad* is the custom JANUS keyboard. Throughout this manual, specific references to the JANUS keyboard use the term *keypad*.
- The *keyboard* buffer is the machine-level buffer that stores key presses and scanned labels. Throughout this manual, specific references to this buffer and its status flags use the term *keyboard*.

Keypad Input

- Keys that you press on the keypad are emphasized in **bold**. For example, "press **Enter**" means you press the key labeled "Enter" on the reader keypad.
- All key names use first-letter capitalization. For example:

  Ctrl = Control key
  Enter = Enter key
  F3 = F3 key

- When you are required to press and release a series of keys in order, the keys are listed in order with no connectors. For example, to enter the uppercase character A, press **Shift A**. To enter this character, you press and release the **Shift** key, and then press the key marked **A**.
- When you are required to press more than one key at the same time, the keys are connected by a dash in the text. For example, press **Ctrl-Alt-Del** to perform a warm boot on a standard PC. When the keys are connected by a dash, it is important that you press and hold the keys in the order they are listed in the text.

Commands

- DOS commands are printed in Courier, exactly as you must type them. For example:

```
COPY INTERMEC.* E:\
```

- Code examples are printed in 8-point Courier. For example:

```
if(step != 0) level = step;    // use step value if provided
if(level >= 31) level = 0;     // keep level within bounds
```

Other Conventions

- *Italic type* identifies a syntax parameter where it is defined in text. Italic type is also used to indicate references to other manuals and to indicate important terminology.
- Hexadecimal numbers in text are followed by an uppercase H. For example, 03 hex is shown as 03H.
- Hexadecimal numbers in C language code segments begin with 0x. For example, `AX_REG = 0x5300`.

Other Intermec Manuals

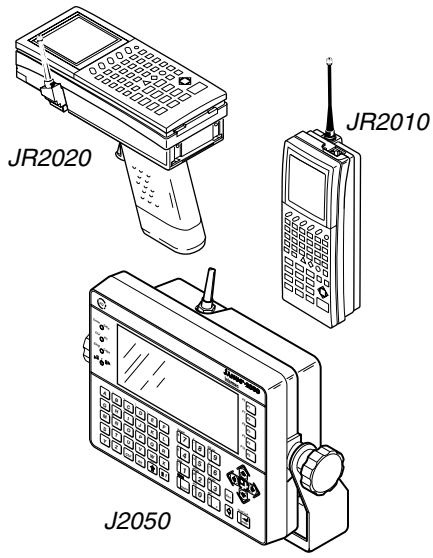
You may need additional information when working with the PSK in a data collection system. Please visit our Web site at www.intermec.com to download many of our current manuals in PDF format. To order printed versions of the Intermec manuals, contact your local Intermec representative or distributor.

1

Getting Started

This chapter briefly describes the programming methods and capabilities that apply to the JANUS family of readers and explains how to install the PSK.

What Your JANUS Reader Can Do



JPSK.001

Your JANUS reader is a portable, programmable bar code reader and a 386-based computer in one. Each reader has the Intermec controlled BIOS and MS-DOS 6.2. The reader behaves and functions like a normal PC with 640K of memory, with these exceptions:

- The JANUS 2010 and 2020 are hand-held, with a small LCD display, custom keyboard, and either a port for a bar code input device or a built-in scanner.
- The JANUS 2050 is mounted to a vehicle, such as a forklift, and has a monochrome CGA display, custom keyboard, built-in RF, and a port for a bar code input device.
- The operating system and protocol support programs reduce the memory available for your application to about 450K.

You can run batch programs and copy, name, or move files in the same manner as you would with any normal PC. The only difference is that the JANUS reader has ROM and RAM drives and a PC card drive.

You can also run DOS-based programs on the reader the same as you can on a PC. When you do, the reader behaves like a PC, but has the advantage of accepting bar code input as if it came from the keypad. If you are an experienced PC programmer, you have already written programs that will run on the reader.

Virtual Wedge

The Virtual Wedge feature of the reader allows application programs to receive decoded bar codes from the keyboard buffer. The Virtual Wedge makes the reader functionally equivalent to a reader wedge connected to a PC. Bar code input is inserted into the PC keyboard buffer as if entered from the keypad. The Virtual Wedge also allows rapid porting of PC applications to the reader.

Note: *If your PC application follows MS-DOS programming conventions, it should run correctly on the JANUS reader. Not all programming languages, especially database languages, follow these conventions. If your PC application does not follow MS-DOS programming conventions, your scanned input will be incorrect.*

Valid configuration commands and reader commands are not put into the keyboard buffer. Bar code configuration commands beginning with $+ are tagged as configuration commands by the Virtual Wedge and sent to the configuration manager to reconfigure the reader. The command parser in the Virtual Wedge software recognizes and processes reader commands.

You can run applications that use the Virtual Wedge (instead of Intermec interrupt extensions or function libraries) on either the reader or on your PC.

Reader Services

Reader services include several functions ranging in complexity from controlling the reader beeper to controlling the function of the power management software. With reader services, you can easily use bar code input from a wand or scanner by using the direct program interface to handle the functions of the reader keyboard, display, and beeper.

There are two methods for incorporating the reader services into your programs:

- Using function libraries
- Using software interrupts

You can use either one or a combination of both methods.

Note: *Do* ***not*** *run programs that use* ***PSK library functions*** *on your PC. If you attempt to run these programs without the Simulator, you will receive an error message and the program will not run.*

Caution
Do not run programs that use **Intermec-specific interrupt extensions** ***on your PC. If you attempt to run these programs without the Simulator, they will cause your PC to lock up and possibly corrupt your system BIOS.***

Consiel
N'exécutez pas de programmes utilisant **des extensions d'interruption spécifiques à Intermec** ***sur votre PC. Si vous tentez de les exécuter sans le Simulator, ces programmes risquent de verrouiller votre PC et d'altérer le BIOS de votre système.***

Function Libraries

You can use a *supported language* and its associated function libraries to access reader services. The term *supported language* describes a language that has extensive Intermec-supplied library functions that control reader services. The supported language functions are stored in libraries contained in the Programmer's Software Kit Language Libraries found at www.intermec.com.

At the time of printing, the list of supported languages includes:

- Borland C/C++
- Microsoft C/C++
- Microsoft Visual C/C++

When you write applications in one of the supported languages, link your program to the library for your chosen language by including the header files in your program, and then invoke the reader services using the language-specific reader service commands. The Intermec-specific functions are documented in Chapter 2, "C Language Support."

Software Interrupts

The reader supports special software interrupts in addition to those available on a normal PC. If your programming language supports software interrupts, you can write applications that use the reader services. However, it is much easier to use the PSK library functions.

Interrupts are a very *low-level* method of controlling a computer. Programming with interrupts is more difficult than with a high-level language (such as C).

Some of the most popular languages have built-in commands for triggering software interrupts (Microsoft C and Borland C++ for example). Languages that do not have built-in interrupt commands sometimes allow you to access interrupts by embedding fragments of assembly language inside your program. There are other methods for triggering software interrupts, and you can generate software interrupts from almost any language.

For complete descriptions and examples of the software interrupts that govern reader services, see Chapter 3, "Software Interrupts," and Appendix B, "Sample Interrupt Programs."

Caution
Do not run programs that use* Intermec-specific interrupt extensions *on your PC. If you attempt to run these programs, they will cause your PC to lock up and possibly corrupt your system BIOS.

Conseil
N'exécutez pas de programmes utilisant* des extensions d'interruption spécifiques à Intermec *sur votre PC. Si vous tentez de les exécuter sans le Simulator, ces programmes risquent de verrouiller votre PC et d'altérer le BIOS de votre système.

Installing the Programmer's Software Kit

The files for the Programmer's Software Kit Language Libraries are distributed in several subdirectories, each corresponding to the supported language:

Subdirectory Name	Language
INTERMEC\BORLANDC	Borland C/C++
INTERMEC\MICROSFT	Microsoft C/C++ and Visual C/C++

The exact contents of the PSK Language Libraries is listed in the README.TXT file. To use the library functions, install the correct files on your computer. You can also use the DOS copy command to copy only the specific files you are sure you will need.

Note: *If you have an existing Intermec directory, the installation process will update any files in the existing Intermec directory with the newer version files having the same name.*

To install the library files

1. Copy the PSK Language Libraries files from www.intermec.com to your computer.
2. Enter the following command:

   `INSTALL` *`dirname drive`*

   where:

   dirname is either BORLANDC, or MICROSFT, depending on which programming language you need.

   drive is the location where you want to install the utilities library. Drive C is the default drive.

2

C Language Support

This chapter describes how to compile and link applications in C using the Programmer's Software Kit (PSK) library functions. It also provides the syntax and parameters for each function.

Building Your Program Using Borland C/C++

Use the following procedure to compile and link a C/C++ program that uses the Borland C/C++ library functions that you installed from the PSK web site at www.intermec.com. You will use one make file and one example program:

XCDEMO.PRJ	Sample make file to create XCDEMO.EXE.
XDEMO.PRJ	Sample make file to create XDEMO.EXE.
XCDEMO.C	Sample reader program using im_receive_input.
XDEMO.C	Sample reader program using IRL-like functions.

If you are building your own program, make sure that you set the following options. The sample programs already include these settings:

- The project must be set to create a DOS-executable file.
- The memory model must be set to the Large compilation model.

To build a sample program using Borland C/C++

1. Start the Borland Programmer's Platform from Microsoft Windows.
2. Open the project file XDEMO.PRJ or XCDEMO.PRJ from:

   `\INTERMEC\BORLANDC\SAMPLES`
3. Open XDEMO.C or XCDEMO.C from:

   `\INTERMEC\BORLANDC\SAMPLES`
4. From the Options menu, set the Directories to the directory containing your Borland C include files and your Borland C library files:

   `\BORLANDC\INCLUDE;\INTERMEC\BORLANDC\INCLUDE`

   `\BORLANDC\LIB;\INTERMEC\BORLANDC\LIB`
5. Compile and link the demo program.

Building Your Program Using Microsoft C/C++

Use the following procedure to compile and link a C/C++ program that uses the Microsoft C/C++ library functions that you installed from the PSK web site at www.intermec.com. You will use one make file and one example program:

XCDEMO.MAK	Sample make file to create XCDEMO.EXE.
XDEMO.MAK	Sample make file to create XDEMO.EXE.
XCDEMO.C	Sample reader program using im_receive_input.
XDEMO.CPP	Sample reader program using IRL-like functions.

If you are building your own program, make sure that you set the following options. The sample programs already include these settings:

- The project must be set to create a DOS-executable file.
- The memory model must be set to the Large compilation model.

To build a sample program using Microsoft C/C++

1. Start the Microsoft Programmer's Workbench from Microsoft Windows.
2. Open XCDEMO.MAK or XDEMO.MAK from:

   `\INTERMEC\MICROSFT\SAMPLES`
3. Open XCDEMO.C or XDEMO.CPP from:

   `\INTERMEC\MICROSFT\SAMPLES`
4. Set OPTIONS | DIRECTORIES to the directory containing your Microsoft C include files and your Microsoft C library files. The sample project uses the following include files and library files:

   `\MICROSFT\INCLUDE;\INTERMEC\MICROSFT\INCLUDE`
   `\MICROSFT\LIB;\INTERMEC\MICROSFT\LIB`
5. Build the demo program from the Project menu.

Debugging a Program With Turbo Debugger

Turbo Debugger is the application debugging environment supplied with Borland C/C++. Since Turbo Debugger requires a lot of memory and is easier to use on a large display, you will probably want to debug your programs using the remote debugging feature. Turbo Debugger runs on your PC while the program you want debugged runs on the reader.

To use remote debugging, you must connect the PC to your reader with a null modem cable and you must run TDREMOTE.EXE on the reader.

To debug your program

1. Connect the PC to your reader with a null modem cable.
2. Copy the TDREMOTE.EXE program to a drive on the reader.
3. If your application uses reader services, make sure they are loaded on the reader. If you use the Intermec-supplied AUTOEXEC.BAT file, they are loaded automatically. Otherwise, start reader services on the reader by entering this command:

   ```
   RSERVICE
   ```

4. On the reader, change to the drive with TDREMOTE.EXE and enter the command:

   ```
   TDREMOTE -rpm -rsn
   ```

   where:

   rp*m* sets the reader's communications port
   m is 1, 2, or 4 for COM1, COM2, or COM4

   rs*n* sets the transmission baud rate as follows:

JANUS Reader	Port	Switch	Baud
2010 and 2050	COM1	-rs2	19,200
		-rs3	38,400
		-rs4	115,000
2020	COM1	-rs2	19,200
		-rs3	38,400
		-rs4	115,000

5. On the PC where Turbo Debugger will run, go to the directory where your program file is located and enter:

   `TD -rp`*`m`* `-rs`*`n`* *`progname`*`.exe`

   where:

rp*m*	sets the PC communications port. Omit this for COM1.
rs*n*	sets the transmission baud rate. This parameter must match the value you entered for -rs*n* on the reader.
progname	is your executable program file.

6. Use Turbo Debugger as described in the user's guide supplied with Borland C/C++ to debug your application.

   Note: *Turbo Debugger uses the communications port of the reader for remote debugging. You need to avoid conflict with portions of your programs that use the reader's communications functions. Skip over the portions that use communications functions.*

Status Code Macros

When using any of the C library functions, you can check for a specific return value or you can use one of the PSK macros to determine the severity of the returned status codes. For portability with future Intermec products, we recommend that you use the macros.

The following macros are located in IMMSG.H:

IM_ISERROR(*status*) This macro returns a nonzero number when *status* indicates an error (either fatal or nonfatal). Zero is returned if *status* indicates either success or warning.

IM_ISSUCCESS(*status*) This macro returns a nonzero number when *status* indicates either success or warning. Zero is returned if *status* indicates an error (either fatal or nonfatal).

IM_ISGOOD(*status*) This macro returns a nonzero number when *status* indicates success.

IM_ISWARN(*status*) This macro returns a nonzero number when *status* indicates a warning.

The following example uses IM_ISERROR to test for an error, and then prints the error message on the reader display.

```
status = im_transmit_byte( com_port, outchar);
if ( IM_ISERROR(status))
im_message(status);
```

Runtime Requirements

For some library functions to work properly, you must install and run specific software components at program execution time. For example, if the reader is using a communications port, you must load a protocol handler. Because protocol handlers use a lot of memory, they are not automatically loaded for you. Other functions require the Reader Wedge TSR program.

Note: *Large C language programs can require more memory than is currently available in the reader. One possible solution is to use overlays. For more information, see your C language programmer's guide.*

Protocol Handlers

You use the Intermec protocol handler (PHIMEC.EXE) when the reader is connected with other Intermec devices. PHIMEC.EXE works with User-Defined, Point-to-Point, Polling Mode D, and Multi-Drop protocols.

You use the PC standard protocol handler (PHPCSTD.EXE) when the reader is connected to devices that use PC standard protocol. PHPCSTD.EXE provides low-level communication abilities and protocol services at the DOS level for non-communication software. It also provides byte-by-byte transfer.

If your reader is a radio frequency (RF) unit, RFPH.EXE is automatically loaded. For more information on protocol handlers, refer to your JANUS user's manual.

To install a protocol handler on the reader

- From the DOS prompt, enter the following command:

  `handler n`

  where:

  handler is either PHIMEC or PHPCSTD.

  n is the number of the communications port.

When your application is finished, the protocol handler TSR continues to run and can prevent other applications from running due to lack of memory. You can unload the protocol handler TSR by adding the unload command as the last line of the batch file that launches your application.

To unload a protocol handler on the reader

- From the DOS prompt, enter the following command:

  `unload` *`handler n`*

  where:

  handler is either PHIMEC or PHPCSTD.

  n is the number of the communications port.

Reader Wedge

Some functions, such as im_receive_input, allow the reader to receive input from multiple sources and process the input depending on the source. You must load the Reader Wedge (RWTSR.EXE) TSR to use these functions.

When your application is finished, the Reader Wedge TSR continues to run in the background and take up memory. The Reader Wedge TSR can prevent other applications from running due to lack of memory. Make sure that your application unloads the Reader Wedge TSR as part of the exit routine or as part of a batch file.

Note: *PSK version 2.1 and above automatically loads RWTSR.EXE for C language applications. However, you get optimal memory management by loading and unloading RWTSR.EXE at the command line. The autoload feature uses about 6K additional memory.*

To load the Reader Wedge TSR on the reader

- From the DOS prompt, enter the following command:

  `RWTSR`

To unload the Reader Wedge TSR on the reader

- From the DOS prompt, enter the following command:

  `RWTSR -D`

Specific Functions With Runtime Requirements

The following table lists the PSK functions that have runtime requirements.

Specific Runtime Requirements

This Function	Requires
im_cancel_rx_buffer	Protocol Handler
im_cancel_tx_buffer	Protocol Handler
im_clear_abort_callback	Reader Wedge
im_command	Reader Wedge
im_get_input_mode	Reader Wedge
im_get_label_symbology	Reader Wedge
im_get_length	Reader Wedge
im_input_status	Reader Wedge
im_irl_a	Reader Wedge
im_irl_k	Reader Wedge
im_irl_n	Reader Wedge
im_irl_v	Reader Wedge If the data is from a communications port, use a Protocol Handler and im_link_comm.
im_irl_y	Reader Wedge Protocol Handler Use im_link_comm.
im_link_comm	Reader Wedge Protocol Handler
im_parse_host_response	Protocol Handler
im_receive_buffer	Protocol Handler Do not use im_link_comm.
im_receive_buffer_no_wait	Protocol Handler Do not use im_link_comm.
im_receive_buffer_noprot	Protocol Handler Do not use im_link_comm.
im_receive_byte	PHPCSTD.EXE You must use PC standard protocol on the host. Do not use im_link_comm.

Specific Runtime Requirements (continued)

This Function	Requires
im_receive_input	Reader Wedge Protocol Handler Use im_link_comm if data is from a communications port.
im_rx_check_status	Protocol Handler
im_serial_protocol_control	Use im_link_comm.
im_set_abort_callback	Reader Wedge
im_set_display_mode	Reader Wedge
im_set_input_mode	Reader Wedge
im_setup_trx	Protocol Handler
im_standard_trx	Protocol Handler
im_standby_wait	Reader Wedge
im_transmit_buffer	Protocol Handler
im_transmit_buffer_no_wait	Protocol Handler
im_transmit_buffer_noprot	Protocol Handler
im_transmit_byte	Protocol Handler You must install PHPCSTD.
im_unlink_comm	Reader Wedge Protocol Handler

Certified C Language Functions

Because of the large number of functions in the C language, not all functions are known to work correctly when used with the JANUS family of readers. Many of these have been tested, however, and are known to operate on the reader the same as on a PC. These tested functions are listed below.

Tested C Functions

atol	_dos_settime	fseek
bdos	dostounix	_fstrlen
_bios_keybrd	enable	fwrite
bioskey	exit	geninterrupt
_bios_serialcom	farcoreleft	getch
_chain_intr	farfree	getchar
chdir	farmalloc	getche
_chdrive	ffeof	getcwd
_chmod	fflush	getdate
_clock	fgets	_getdcwd
chsize	filelength	getdfree
close	fileno	getdisk
clrscn	findfirst	_getdrive
corel	findnext	getenv
cprintf	fnmerge	getpsp
cputs	fnsplit	gets
delay	fopen	gettext
disable	fprintf	gettextinfo
_dos_findnext	fputc	gettime
_dos_getdate	fputs	getvect
_dos_getdiskfree	FP_OFF	gotoxy
_dos_getfileattr	FP_SEG	_harderr
_dos_gettime	fread	_hardresume
_dos_setdate	free	_hardretn

Tested C Functions (continued)

highvideo	qsort	va_start
inport	read	vsprintf
inportb	realloc	wherex
inpw	remove	wherey
int86	scanf	window
intdosx	setcursortype	write
isalpha	setdate	
isascii	settime	
iscntrl	setvect	
isdigit	sleep	
isprint	sprintf	
isspace	sscanf	
itoa	strcat	
kbhit	strchr	
keep	strcmp	
lseek	strcmpi	
malloc	strcpy	
memccpy	strcspn	
memcmp	strlen	
memcpy	strncmp	
memmove	strncmpi	
memset	strncpy	
min	strnset	
MK_FP	strpbrk	
open	strrchr	
outp	strstr	
outportb	tell	
outpw	textbackground	
printf	textcolor	
putch	time	
puts	toupper	
puttext	va_end	

Intermec C Library Functions

The following pages list the C library functions available in the Programmer's Software Kit (PSK), along with notes on syntax and examples of how the functions can be used. Each function is fully compatible with currently available Microsoft C/C++ and Borland C/C++ compilers.

Note: *Do* ***not*** *run programs that use* ***PSK library functions*** *on your PC. If you attempt to run these programs without the Simulator, you will receive an error message and the program will not run.*

Caution
Do not run programs that use **Intermec-specific interrupt extensions** ***on your PC. If you attempt to run these programs, they will cause your PC to lock up and possibly corrupt your system BIOS.***

Conseil
N'exécutez pas de programmes utilisant **des extensions d'interruption spécifiques à Intermec** ***sur votre PC. Si vous tentez de les exécuter sans le Simulator, ces programmes risquent de verrouiller votre PC et d'altérer le BIOS de votre système.***

C Library Functions Listed by Category

Communications

DCM

Display

Input

IRL

Keypad

Macros

Program Control

Sound

System

Viewport

Note: *The following syntax descriptions refer to many named constant variables, such as IM_COM1. These variables always appear in uppercase and are described in IM20LIB.H.*

im_appl_break_status

Purpose: This function checks whether the application break sequence has been pressed. All PSK functions are terminated when the application break sequence is keyed in.

Place calls to this function at strategic locations in your program to detect when a user wants to break out of a loop. Your program can determine what action to take when the break sequence is detected.

Syntax:

```
#include <im20lib.h>
IM_STATUS im_appl_break_status
    (IM_APPBREAK_STATUS *brk);
```

IN Parameters: None.

OUT Parameters: The *brk* parameter returns a nonzero value if the break status bit is set and zero if it is not set.

Return Value: This function returns one of the standard status codes defined in Appendix A, "Status Codes."

Notes: The *brk* flag is reset each time the routine is called. Because this function is also called during the input routines provided in the library, you should check the return code from input functions for the hex value 8515H. This value indicates that the input function has terminated because the application break sequence was entered. In this case, *brk* is cleared by the input function's call to im_appl_break_status.

For more information on the application break sequence, refer to your JANUS user's manual.

To enter an application break sequence

1. Press (I/O) to turn off the reader.
2. Press (F3) - 2 - ◆ at the same time.
3. Press 1 . This sets the reader's application break bit.
4. Press (I/O) to turn on the reader.

Example

```
/******************* im_appl_break_status ****************************/
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include "im20lib.h"

void main (void)
{
IM_STATUS          status;
IM_APPBREAK_STATUS brk;

   /* Pause for key stroke */
   while ( !kbhit() )
      ;
   /* To set the break bit: */
   /* Press the following sequence at this pause: */
   /* I/O  key (Turn OFF the reader)              */
   /* F3+2+Left Arrow (Press F3, 2, and left arrow at the same time) */
   /* 1    (Press the 1 (one) key to set application break bit) */
   /* I/O  key (Turn the reader ON) */
   /* Enter key */

   /* Leave the break bit unset: Press any key. */

   /* Set brk to known value */
   brk = 99;

   /* Call Intermec function.  Everytime the break bit is checked, it is cleared*/
   status = im_appl_break_status ( &brk);

   if (brk)
   {
      /* brk should return a -1 if bit is set */
      printf ("Break status ON\n");
      printf ("return status: %4X\n", status);
      printf ("break status: %d\n", brk);
   }
   else
   {
      /* brk should return a 0 if bit is not set */
      printf ("Break status OFF\n");
      printf ("return status: %4X\n", status);
      printf ("break status: %d\n", brk);
   }
}
```

Note: Break *is reserved. So, do not name the application Break.cpp.*

im_backlight_off

Purpose: This function turns the display backlight off. Turn off the backlight to prolong the reader's battery life.

Syntax:
```
#include <im20lib.h>
void im_backlight_off(void);
```

IN Parameters: None.

OUT Parameters: None.

Return Value: None.

Notes: You can program the backlight to automatically turn off after a specified time to save battery power. The length of time depends on the backlight timeout configuration parameter. The configuration parameter is set in 1-second increments. The default is 10 seconds.

This function has no effect on the JANUS 2050.

See Also: im_backlight_on, im_backlight_toggle

Example

```
/******************* im_backlight_off ****************************/
#include <stdio.h>
#include "im20lib.h"

void main (void)
{
   im_backlight_off();
   printf("Backlight off\n");
}
```

im_backlight_on

Purpose: This function turns the display backlight on to illuminate the reader display in dimly lit environments.

Syntax:
```
#include <im20lib.h>
void im_backlight_on(void);
```

IN Parameters: None.

OUT Parameters: None.

Return Value: None.

Notes: You can program the backlight to automatically turn off after a specified time to save battery power. The length of time depends on the backlight timeout configuration parameter. The configuration parameter is set in 1-second increments. The default is 10 seconds.

This function has no effect on the JANUS 2050.

See Also: im_backlight_off, im_backlight_toggle

Example

```
/******************* im_backlight_on ***************************/
#include <stdio.h>
#include "im20lib.h"

void main (void)
{
   im_backlight_on();
   printf("Backlight on\n");
}
```

im_backlight_toggle

Purpose: This function toggles the display backlight ON and OFF.

Syntax:
```
#include <im20lib.h>
void im_backlight_toggle(void);
```

IN Parameters: None.

OUT Parameters: None.

Return Value: None.

Notes: You can program the backlight to automatically turn off after a specified time to save battery power. The length of time depends on the backlight timeout configuration parameter. The configuration parameter is set in 1-second increments. The default is 10 seconds.

This function has no effect on the JANUS 2050.

See Also: im_backlight_on, im_backlight_off

Example

```
/******************* im_backlight_toggle ****************************/
#include <stdio.h>
#include "im20lib.h"

void main (void)
{
   im_backlight_toggle();
   printf("Backlight switched on/off\n");
}
```

im_cancel_rx_buffer

Purpose: This function clears the receive buffer of the designated communications port.

Syntax:
```
#include <im20lib.h>
IM_STATUS im_cancel_rx_buffer
    (IM_COM_PORT port_id);
```

IN Parameters: The *port_id* parameter identifies the communications port as follows:

IM_COM1	COM1
IM_COM2	COM2, scanner port (2010 and 2050)
IM_COM4	COM4 (RF only)

OUT Parameters: None.

Return Value: This function returns one of the standard status codes defined in Appendix A, "Status Codes."

If there is no input pending to receive, this function returns 4602H (no client).

Notes: You must install a protocol handler to use this function.

See Also: im_receive_buffer_no_wait, im_receive_buffer_noprot

Example

```
See example for im_receive_buffer_no_wait.
```

im_cancel_tx_buffer

Purpose: This function clears the contents of the transmit buffer for a designated communications port, stops transmission in progress, and resets the communications port.

Use this function with Polling Mode D to clear out the buffer. Other protocols clear the buffer automatically.

You can also use this function to clear a busy port.

Syntax:

```
#include <im20lib.h>
IM_STATUS im_cancel_tx_buffer
    (IM_COM_PORT port_id);
```

IN Parameters: The *port_id* parameter identifies the communications port as follows:

IM_COM1	COM1
IM_COM2	COM2, scanner port (2010 and 2050)
IM_COM4	COM4 (RF only)

OUT Parameters: None.

Return Value: This function returns one of the standard status codes defined in Appendix A, "Status Codes."

Notes: You must install a protocol handler to use this function.

See Also: im_transmit_buffer_no_wait, im_transmit_buffer_noprot, im_transmit_buffer

Example

```
See example for im_transmit_buffer_no_wait.
```

im_clear_abort_callback

Purpose: This function clears the notification (callback) mode of the Virtual Wedge.

Syntax:
```
#include <im20lib.h>
IM_STATUS im_clear_abort_callback(void);
```

IN Parameters: None.

OUT Parameters: None.

Return Value: This function returns one of the standard status codes defined in Appendix A, “Status Codes.”

Notes: You must install the Reader Wedge to use this function. For information on loading RWTSR.EXE, see “Runtime Requirements” earlier in this chapter.

See Also: im_set_abort_callback

Example

```
See example for im_set_abort_callback.
```

im_command

Purpose: This function modifies the reader's configuration. For example, you can use this function to set a specific communications port's baud rate.

Syntax:

```
#include <im20lib.h>
IM_STATUS im_command
    (IM_UCHAR *command,
    IM_USHORT command_length);
```

IN Parameters: The *command* parameter is a reader command string. The command string may include more than one reader command. For example, the command string %.1$+BV9 turns on the backlight and raises the beep volume.

Reader commands are listed in your JANUS user's manual.

The *command_length* parameter is the length of the reader command string.

OUT Parameters: None.

Return Value: This function returns one of the standard status codes defined in Appendix A, "Status Codes."

Notes: For more information on using im_command, see your JANUS user's manual.

You must install the Reader Wedge to use this function. For information on loading RWTSR.EXE, see "Runtime Requirements" earlier in this chapter.

Example

```
/******************* im_command ************************************/
/* Also with: im_standby_wait */

#include <stdio.h>
#include <string.h>
#include "im20lib.h"

void main (void)
{
   /* Command string to set contrast */
   char   *high_trast   ="$+DJ7";       /* 7 (dark) */
   char   *normal_trast ="$+DJ3";
   char   *low_trast    ="$+DJ0";       /* 0 (light)*/

   /* Set high contrast */
   printf("Set high contrast\n");
   im_command(high_trast, strlen(high_trast));

   /* Wait for two seconds */
   im_standby_wait(2000);

   /* Set low contrast */
   printf("Set low contrast\n");
   im_command(low_trast, strlen(low_trast));

   /* Wait for two seconds */
   im_standby_wait(2000);

   /* Set normal contrast */
   printf("Set normal contrast\n");
   im_command(normal_trast, strlen(normal_trast));
}
```

im_cursor_to_viewport

Purpose: This function moves the cursor to the center of the current viewport. Since the viewport can move around with or without the cursor, you can use this function to recenter the cursor.

Syntax:
```
#include <im20lib.h>
void im_cursor_to_viewport(void);
```

IN Parameters: None.

OUT Parameters: None.

Return Value: None.

Notes: This function has no effect on the JANUS 2050.

See Also: im_viewport_end, im_viewport_home, im_viewport_page_down, im_viewport_page_up, im_viewport_to_cursor, im_viewport_move

Example

```
See example for im_viewport_setxy.
```

im_decrease_contrast

Purpose: This function decreases the LCD display contrast by one level. The display contrast is the lightness or darkness of the characters against the reader display. The reader display has 32 levels of contrast (0 to 31), where 0 is very light, and 31 is very dark.

Syntax:

```
#include <im20lib.h>
IM_STATUS im_decrease_contrast(void);
```

IN Parameters: None.

OUT Parameters: None.

Return Value: This function returns one of the standard status codes defined in Appendix A, “Status Codes.”

Notes: The 0 to 31 scale fine tunes the contrast more precisely than using the keypad to adjust the contrast.

The functions im_set_contrast and im_get_contrast use a scale of 0 to 7 that is related to the scale of 0 to 31 used by im_increase_contrast and im_decrease_contrast. The following table compares the two scales.

This Value on Scale 0 to 7	Equates to This Value on Scale 0 to 31	Contrast Level
0	10	Very light
1	12	
2	14	
3	16	
4	18	
5	20	
6	22	
7	24	Very dark

This function has no effect on the JANUS 2050.

See Also: im_increase_contrast, im_set_contrast, im_get_contrast

Example

```
/******************* im_decrease_contrast ****************************/
#include <stdio.h>
#include "im20lib.h"
#include "immsg.h"

void main (void)
{
IM_STATUS status;

   status = im_decrease_contrast();

   /* If the most significant bit is set, there is an error */
   if (IM_ISERROR(status))
      printf("Dec Contrast error = %x\n", status);
}
```

im_get_config_info

Purpose: This function retrieves the current reader configuration information string and its length.

Syntax:

```
#include "im20lib.h"
IM_STATUS im_get_config_info
    (IM_UCHAR *config_string,
    IM_USHORT *length);
```

IN/OUT Parameters: The *config_string* parameter is the configuration information string. The first two characters specify the type of configuration information returned.

For example, to get the beep duration setting, set *config_string* to "BD." The function returns BD and the current configuration for beep duration.

The *length* parameter is the length of the configuration information string.

For a list of the configuration commands, see your JANUS user's manual.

Return Value: This function returns one of the standard status codes defined in Appendix A, "Status Codes."

Notes: This function differs from im_command in that you only pass the two-character command identifier. The im_command function passes an entire command string.

See Also: im_command

Example

```
/******************* im_get_config_info ****************************/
#include <stdio.h>
#include <string.h>
#include "im20lib.h"


void main(void)
{
   IM_STATUS status;
   IM_USHORT length;
   IM_UCHAR  config_string[128];

// Use im_get_config_info to get the Beeper Volume
   printf("\nim_get_config_info example: \n");

// Set config request for Beeper Volume
   strcpy(config_string, "BV");

   status = im_get_config_info(config_string, &length);

// Print the results
   printf("\nBeeper Volume: %s", config_string);
   printf("\nlength: %d", length);
   printf("\nstatus: %d", status);
}
```

im_get_contrast

Purpose: This function gets the reader's LCD display contrast level. There are eight levels of contrast (0 to 7) from very light to very dark, which are the most frequently used contrast settings.

Syntax:

```
#include "im20lib.h"
IM_STATUS im_get_contrast
    (IM_UCHAR *display_contrast_level);
```

IN Parameters: None.

OUT Parameters: The *contrast* parameter is a number from 0 to 7 or is one of these constants:

IM_MIN_CONTRAST	0	Minimum contrast
IM_MAX_CONTRAST	1	Maximum contrast

Return Value: This function returns one of the standard status codes defined in Appendix A, "Status Codes."

Notes: The functions im_set_contrast and im_get_contrast use a scale of 0 to 7 that is related to the scale of 0 to 31 used by im_increase_contrast and im_decrease_contrast. The following table compares the two scales.

This Value on Scale 0 to 7	Equates to This Value on Scale 0 to 31	Contrast Level
0	10	Very light
1	12	
2	14	
3	16	
4	18	
5	20	
6	22	
7	24	Very dark

This function has no effect on the JANUS 2050.

See Also: im_set_contrast, im_decrease_contrast, im_increase_contrast

Example

```
See example for im_set_contrast.
```

im_get_control_key

Purpose: You can enable or disable the **Ctrl** key. This procedure retrieves the current setting.

When you disable this keycode, none of the key combinations that use the **Ctrl** key will work. For example, the warm boot key sequence **Ctrl-Alt-Del** will not work.

Syntax:

```
#include "im20lib.h"
IM_STATUS im_get_control_key
    (IM_CONTROL *control_key);
```

IN Parameters: None.

OUT Parameters: The *control_key* parameter is one of these constants:

IM_ENABLE	**Ctrl** key is enabled
IM_DISABLE	**Ctrl** key is disabled

Return Value: This function returns one of the standard status codes defined in Appendix A, “Status Codes.”

See Also: im_set_control_key

Example

```
/******************* im_get_control_key ******************************/
/* Also with: im_set_control_key */

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <ctype.h>
#include "im20lib.h"

void main (void)
{
IM_UCHAR    user_option;
IM_CONTROL  old_controlkey;
IM_CONTROL  new_controlkey;

   clrscr();                  /* Clear screen */
   gotoxy(1,1);               /* Position message */

   /* Get the current status of control key */
   im_get_control_key(&old_controlkey);

   /* Display control key's status */
   if (old_controlkey == IM_ENABLE)
      printf("<CTRL> key enabled!");
   else
      printf("<CTRL> key disabled!");

   /* Get user's input option, 'y' for enabling control key */
   printf("\n\nEnabling <CTRL> key(y)?");
   user_option =  getche();
   user_option = toupper(user_option);

   if (user_option == 'Y')
   {
      /* <CTRL> key enable */
      im_set_control_key(IM_ENABLE);
   }
   else
   {
      /* <CTRL> key disable */
      im_set_control_key(IM_DISABLE);
   }

   /* Get  new <CTRL> key's status after setting */
   im_get_control_key(&new_controlkey);

   /* Display user's input setting */
   if (new_controlkey == IM_ENABLE)
      printf("\n<CTRL> key enabled!");
   else
      printf("\n<CTRL> key disabled!");

   printf("\nTest it>>\n");
}
```

im_get_display_mode

Purpose: This function gets the size mode, video mode, scroll mode, and character height of the display.

Syntax:

```
#include "im20lib.h"
IM_STATUS im_get_display_mode
    (IM_STD_SIZE_MODE *size_mode,
    IM_STD_VIDEO_MODE *video_mode,
    IM_SCROLL_MODE *scroll_mode,
    IM_CHARACTER_HEIGHT *char_ht);
```

IN Parameters: None.

OUT Parameters: The *size_mode* parameter is required and is one of these constants:

IM_SIZE_MODE_80X25	80 x 25 text mode
IM_SIZE_MODE_20X16	20 x 16 text mode (2010 and 2020)
IM_SIZE_MODE_20X8	20 x 8 text mode (2010 and 2020)
IM_SIZE_MODE_10X16	10 x 16 text mode (2010 and 2020)
IM_SIZE_MODE_10X8	10 x 8 text mode (2010 and 2020)

If the *size_mode* parameter is set to IM_SIZE_MODE_80X25, the video mode, scroll mode, and character height values may have been customized. If any other *size_mode* is set, the video mode, scroll mode, and character height use the default values.

The *video_mode* parameter requires IM_SIZE_MODE_80X25. The standard PC BIOS supports up to video mode 13H. The reader only supports up to video mode 6. You can also set video modes 0 through 3 with IC.EXE.

The *video_mode* parameter is one of these constants:

IM_STD_VIDEO_MODE_0	40 x 25 (use for double-wide character)
IM_STD_VIDEO_MODE_1	40 x 25 (use for double-wide character)
IM_STD_VIDEO_MODE_2	80 x 25
IM_STD_VIDEO_MODE_3	80 x 25
IM_STD_VIDEO_MODE_4	300 x 200 2-color
IM_STD_VIDEO_MODE_5	300 x 200 monochrome
IM_STD_VIDEO_MODE_6	640 x 200 2-color (2050)

The *scroll_mode* requires IM_SIZE_MODE_80X25 and is one of these constants:

IM_LCD_SCROLL_AT_25	Scroll at line 25
IM_LCD_SCROLL_AT_16	Scroll at line 16 (2010 and 2020)
IM_LCD_SCROLL_AT_13	Scroll at line 13 (2050 and double height)
IM_LCD_SCROLL_AT_8	Scroll at line 8 (2010 and 2020)

The *char_ht* parameter requires IM_SIZE_MODE_80X25 and is one of these constants:

IM_STANDARD_CHAR_HEIGHT	Standard height characters. On a JANUS 2050, up to 25 lines of these characters fit on the screen. On other JANUS readers, up to 16 lines fit on the screen.
IM_DOUBLE_CHAR_HEIGHT	Double height characters. On a JANUS 2050, up to 13 lines of these characters fit on the screen. On other JANUS readers, up to 8 lines fit on the screen.

Return Value: This function returns one of the standard status codes defined in Appendix A, “Status Codes.”

See Also: im_set_display_mode

Example

```
See example for im_viewport_setxy.
```

im_get_display_type

Purpose: This function gets the hardware display type, either standard JANUS reader or JANUS 2050 (VMU).

Syntax:
```
#include "im20lib.h"
IM_STATUS im_get_display_type
    (IM_DISPLAY_TYPE *type);
```

IN Parameters: None.

OUT Parameters: The *type* parameter is one of these constants:

IM_LCD_20X16	Standard JANUS display
IM_CRT_80X25	JANUS VMU display

Return Value: This function returns one of the standard status codes defined in Appendix A, "Status Codes."

See Also: im_get_display_mode, im_set_display_mode

Example

```
/******************* im_get_display_type ******************************/
#include <stdio.h>
#include <string.h>
#include "im20lib.h"


void main(void)
{
   IM_DISPLAY_TYPE type;
   IM_STATUS status;

// Use im_get_display_type to get the Display Type
   printf("\nim_get_display_type example: \n");

   status = im_get_display_type(&type);

// Print the results
   if(type == IM_LCD_20X16)
   // Standard JANUS Display
      printf("\nDisplay type is Standard JANUS display");
   else
   if(type == IM_CRT_80X25)
   // JANUS VMU Display
      printf("\nDisplay type is JANUS VMU display");
}
```

im_get_follow_cursor

Purpose: When the display is in 80 x 25 mode, the viewport follows the cursor as it moves. The follow-the-cursor feature can be enabled or disabled. This function retrieves the current setting.

Syntax:

```
#include "im20lib.h"
IM_STATUS im_get_follow_cursor
    (IM_CONTROL *follow_cursor);
```

IN Parameters: None.

OUT Parameters: The *follow_cursor* parameter is one of these constants:

IM_ENABLE	Follow-the-cursor mode is enabled
IM_DISABLE	Follow-the-cursor mode is disabled

Return Value: This function returns one of the standard status codes defined in Appendix A, “Status Codes.”

Notes: This function has no effect on the JANUS 2050.

See Also: im_set_follow_cursor, im_cursor_to_viewport, im_viewport_to_cursor

Example

```
/******************* im_get_follow_cursor ******************************/
/* Also with: im_set_follow_cursor */

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <ctype.h>
#include "im20lib.h"

#define  ESC_CHAR      0x1B

void main (void)
{
IM_UCHAR    ch,
            user_option;
IM_CONTROL  old_lock_cursor;
IM_CONTROL  new_lock_cursor;

   clrscr();                  /* Clear screen */
   gotoxy(1,1);               /* Position message */

   /* Get the current status of viewport follow cursor for restore later */
   im_get_follow_cursor(&old_lock_cursor);

   /* Get user's input option, 'y' for disable viewport follow cursor */
   printf("Enable viewport\nfollows cursor(y)?");
   user_option =  getche();
   user_option = toupper(user_option);

   if (user_option == 'Y')
   {
      /* Enable viewport follow cursor */
      im_set_follow_cursor(IM_ENABLE);
   }
   else
   {
      /* Disable viewport follow cursor */
      im_set_follow_cursor(IM_DISABLE);
   }

   /* Get new viewport follow cursor status after setting */
   im_get_follow_cursor(&new_lock_cursor);

   /* Display user's input message */
   printf("\n\n'ESC' to quit!\nType keys to check\n");
   printf("viewport follow\n");

   if (new_lock_cursor == IM_ENABLE)
      printf("cursor ON !!!!!");
   else
      printf("cursor OFF !!!!");

   /* Enter any characters to check viewport follow cursor until 'ESC' key. */
   while ( (ch = getche()) != ESC_CHAR )
       ;

   /* Restore original follow cursor setting */
   im_set_follow_cursor(old_lock_cursor);
}
```

im_get_input_mode

Purpose: This function reports the current mode of the input manager. The different modes affect how the reader interprets and stores input.

Syntax:
```
#include <im20lib.h>
IM_MODE im_get_input_mode (void);
```

IN Parameters: None.

OUT Parameters: None.

Return Value: The IM_MODE returned is one of these constants:

IM_WEDGE	Wedge mode
IM_PROGRAMMER	Programmer mode
IM_DESKTOP	Desktop mode

Notes: There are three different reader input modes: Wedge, Programmer, and Desktop. These different modes affect how the reader interprets and stores input.

Wedge Mode Keypad and label inputs go into the keyboard buffer with minimal filtering. Wedge mode is the default mode at the DOS prompt after reader services (RSERVICE.EXE) is loaded. Use Wedge mode when the reader is running an application that uses the Virtual Wedge. When in this mode, use standard input functions to retrieve keypad or label input.

Wedge mode does not take full advantage of the reader's power management capabilities. You will probably notice reduced battery life when the reader is in this mode compared to either Programmer or Desktop mode. For more information on power management, see Chapter 4, "Advanced Programming."

Programmer Mode Inputs are echoed to the screen. Reader commands are executed and saved. Use Programmer mode when the reader is executing IRL commands. In general, when you are running an application that uses the function libraries or software interrupts, the reader should be in Programmer mode.

Desktop Mode The application is responsible for retrieving and displaying input. Use Desktop mode when you need detailed information about a pressed key. Each character returned consists of four bytes: the ASCII code, scan code, and two bytes for the keyboard flags (**Shift**, **Ctrl**, **Alt**). See the structure IM_KEYCODE in IM20LIB.H for further details.

For more information about reader input modes, see Chapter 4, "Advanced Programming."

You must install the Reader Wedge to use this function. For information on loading RWTSR.EXE, see "Runtime Requirements" earlier in this chapter.

See Also: im_set_input_mode

Example

```
See example for im_receive_input.
```

im_get_keyclick

Purpose: You can configure the reader to emit a click each time a key is pressed. This function returns the current setting (enabled or disabled) of the keyclick.

Syntax:

```
#include "im20lib.h"
IM_STATUS im_get_keyclick
    (IM_CONTROL *keyclick);
```

IN Parameters: None.

OUT Parameters: The *keyclick* parameter is one of these constants:

IM_ENABLE	Keyclick is enabled
IM_DISABLE	Keyclick is disabled

Return Value: This function returns one of the standard status codes defined in Appendix A, “Status Codes.”

See Also: im_set_keyclick

Example

```
/******************* im_get_keyclick ******************************/
/* Also with: im_set_keyclick */

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <ctype.h>
#include "im20lib.h"

void main (void)
{
IM_UCHAR    user_option;
IM_CONTROL  old_keyclick;
IM_CONTROL  new_keyclick;

   clrscr();                  /* Clear screen */
   gotoxy(1,1);               /* Position message */

   /* Get the current status of keypad clicker */
   im_get_keyclick(&old_keyclick);

   /* Display keypad clicker's status */
   if (old_keyclick == IM_ENABLE)
      printf("Keypad clicker ON!");
   else
      printf("Keypad clicker OFF!");

   /* Get user's input option, 'y' for enabling keypad clicker */
   printf("\n\nEnabling keyclick(y)?");
   user_option =  getche();
   user_option = toupper(user_option);

   if (user_option == 'Y')
   {
      /* Keypad clicker enable */
      im_set_keyclick(IM_ENABLE);
   }
   else
   {
      /* Keypad clicker disable */
      im_set_keyclick(IM_DISABLE);
   }

   /* Get new keypad clicker's status after setting */
   im_get_keyclick(&new_keyclick);

   /* Display user's input setting */
   if (new_keyclick == IM_ENABLE)
      printf("\nKeypad clicker ON!");
   else
      printf("\nKeypad clicker OFF!");

   printf("\nTest it>>\n");
}
```

im_get_label_symbology

Purpose: This function gets the symbology, such as Code 39, from the most recently scanned label. Call this function after receiving the data using im_receive_input.

Syntax:

```
#include "im20lib.h"
IM_STATUS im_get_label_symbology
    (IM_DECTYPE *symbology);
```

IN Parameters: None.

OUT Parameters: The *symbology* parameter is one of these constants:

IM_UNKNOWN_DECODE	Unknown bar code
IM_CODABAR	Codebar bar code
IM_CODE_11	Code 11 bar code
IM_CODE_16K	Code 16K bar code
IM_CODE_39	Code 39 bar code
IM_CODE_49	Code 93 bar code
IM_CODE_93	Code 49 bar code
IM_CODE_128	Code 128 bar code
IM_I_2_OF_5	Interleaved 2 of 5
IM_MSI	MSI bar code
IM_PDF_417	PDF417 bar code
IM_PLESSEY	Plessey bar code
IM_UPC	Universal Product Code

Return Value: This function returns one of the standard status codes defined in Appendix A, “Status Codes.”

Notes: You must install the Reader Wedge to use this function. For information on loading RWTSR.EXE, see “Runtime Requirements” earlier in this chapter.

See Also: im_receive_input

Example

```
/******************* im_get_label_symbology ***********************/
/* im_receive_input */
/* im_set_input_mode  */

#include <stdio.h>
#include <conio.h>
#include "im20lib.h"
#include "immsg.h"

static char *bar_code[] = {
  "unknown",
  "Code 39",          /* See typedef enum { ...} IM_DECTYPE in im20lib.h*/
  "Code 93",
  "Code 49",
  "I 2 of 5",
  "Codabar",
  "UPC and EAN",
  "Code 128",
  "Code 16K",
  "Plessey/Anker",
  "Code 11",
  "MSI",
  "PDF417"
};

void main (void)
{
   IM_UCHAR   input[256];
   IM_ORIGIN  source;
   IM_DECTYPE symbol;

   window(1, 1, 20, 16);       /* Set the application text window */
   clrscr();                   /* Clear the screen  */

   printf("Demo im_get_label_symbology\n'q' to quit\n");

   /* Set Reader Wedge into PROGRAMMER mode  */
   im_set_input_mode(IM_PROGRAMMER);

   /* Input loop  */
   do
   {
      source = IM_LABEL_SELECT | IM_KEYBOARD_SELECT;

      /* Request input from Reader Wedge */
      im_receive_input(source, IM_INFINITE_TIMEOUT, &source, input);

      /* Get most recently read label input                    */
      /* im_get_label_symbology only works with im_receive_input */
      im_get_label_symbology( &symbol);

      printf("\n%s\n", input);          /* Display input data */

      /* Display symbology */
      printf("\nSYMBOLOGY: %d\n%s\n", symbol, bar_code[symbol]);

   } while (input[0] != 'q' && input[0] != 'Q');  /* 'q' to quit */

   /* Switch the Reader Wedge back to virtual wedge mode */
   im_set_input_mode(IM_WEDGE);
}
```

im_get_length

Purpose: This function returns the length of the string received from the designated source by the most recent im_receive_input function.

Syntax:
```
#include "im20lib.h"
IM_USHORT im_get_length
    (IM_ORIGIN source);
```

IN Parameters: The *source* parameter is one of these constants:

IM_LABEL_SELECT	Label selected
IM_KEYBOARD_SELECT	Keypad selected
IM_COM1_SELECT	COM1 selected
IM_COM2_SELECT	COM2, scanner port selected (2010 and 2050)
IM_COM4_SELECT	COM4 selected (RF only)

OUT Parameters: None.

Return Value: This function returns the length of the last input string read from the designated source.

Notes: All input using im_receive_input has a null termination character added to the end of the string so that it can be used as a normal C string. However, some data might contain embedded null characters. If so, this function supplies the true data length.

For this function, all COM input is considered to be from the same source. For example, if you call im_receive_input with IM_COM1_SELECT and again with IM_COM4_SELECT, then a call to im_get_length with a source of IM_COM1_SELECT returns the length of the message received from COM4 because that was the last message read from a communications port.

You must install the Reader Wedge to use this function. For information on loading RWTSR.EXE, see “Runtime Requirements” earlier in this chapter.

See Also: im_receive_input

Example

```
See example for im_receive_input.
```

im_get_postamble

Purpose: This function retrieves the currently configured postamble string and its length.

Syntax:
```
#include "im20lib.h"
IM_STATUS im_get_postamble
    (IM_UCHAR *post_string,
    IM_USHORT *length);
```

IN Parameters: None.

OUT Parameters: The *post_string* parameter is the postamble string.

The *length* parameter is the length of the postamble string.

Return Value: This function returns one of the standard status codes defined in Appendix A, "Status Codes."

Notes: You can set the postamble with IC.EXE, with the im_command function and $+AE command, or by scanning a postamble. Refer to your JANUS user's manual for more information on configuring a postamble.

See Also: im_get_postamble, im_get_config_info

Example

```
/******************* im_get_postamble ******************************/

#include <stdio.h>
#include <string.h>
#include "im20lib.h"

void main(void)
{
   IM_STATUS status;
   IM_USHORT length;
   IM_UCHAR  post_string[128];

// Use im_get_postamble to get the Postamble string
   printf("\nim_get_postamble example: \n");

   status = im_get_postamble(post_string, &length);

// Print the results
   printf("\nPostamble: %s", post_string);
   printf("\nlength   : %d", length);
   printf("\nstatus   : %d\n", status);
}
```

im_get_preamble

Purpose: This function retrieves the currently configured preamble string and its length.

Syntax:

```
#include "im20lib.h"
IM_STATUS im_get_preamble
    (IM_UCHAR *pre_string,
    IM_USHORT *length);
```

IN Parameters: None.

OUT Parameters: The *pre_string* parameter is the preamble string.

The *length* parameter is the length of the preamble string.

Return Value: This function returns one of the standard status codes defined in Appendix A, "Status Codes."

Notes: You can set the preamble with IC.EXE, with the im_command function and $+AE command, or by scanning a preamble. Refer to your JANUS user's manual for more information on configuring a preamble.

See Also: im_get_postamble, im_get_config_info

Example

```
/******************* im_get_preamble ******************************/
#include <stdio.h>
#include <string.h>
#include "im20lib.h"

void main(void)
{
   IM_STATUS status;
   IM_USHORT length;
   IM_UCHAR  pre_string[128];

// Use im_get_preamble to get the Preamble string
   printf("\nim_get_preamble example: \n");

   status = im_get_preamble(pre_string, &length);

// Print the results
   printf("\nPreamble: %s", pre_string);
   printf("\nlength  : %d", length);
   printf("\nstatus  : %d", status);
}
```

im_get_reboot_flag

Purpose: This function determines if the reader received a “Prepare For Reboot” command (..+%1) from a host for binary file transfer (BFT). Your application should periodically check if it has received a “Prepare for Reboot” command.

Syntax:
```
#include "im20lib.h"
IM_USHORT im_get_reboot_flag( )
```

IN Parameters: None.

OUT Parameters: None.

Return Value: This function returns one of the following:

1 “Prepare for Reboot” (..+%1) has been received from the host.

0 “Prepare for Reboot” (..+%1) has not been received from the host.
or
“Cancel Reboot Preparation” (..+%0) has been received from the host.

Notes: When a host initiates a BFT, the JANUS reader must reboot with a server session open to establish a transfer session. The application should close any open files and perform cleanup tasks before rebooting.

Input functions such as im_receive_input terminate when the “Prepare for Reboot” command is received. The application should check the return status from input functions to determine if “Prepare for Reboot” terminated the input function.

See Also: im_ready_for_reboot

Example

```
//******************im_get_reboot_flag***********************************
#include <conio.h>
#include <stdio.h>
#include "im20lib.h"
void main (void)
{
    char       inString[300];          // Buffer used by application
    char       comString[300];         // Buffer used by comm utilities
    int        done = 0;
    int        length;
    int        valid_source;
    IM_USHORT  status;
    IM_USHORT  reboot_flag;
    IM_ORIGIN  source;
    IM_DECTYPE symbology;

    /* FUNCTION BODY */

    clrscr();
    im_set_input_mode(IM_PROGRAMMER);
    im_set_display_mode(IM_SIZE_MODE_20X16,
                        IM_STD_VIDEO_MODE_3,
                        IM_LCD_SCROLL_AT_16,
                        IM_STANDARD_CHAR_HEIGHT);
    im_link_comm(IM_COM1, comString, 300);
    while (!done) {
         valid_source = 1;      // assume source will be valid
         printf("\nEnter>>");
         printf("\nPress Q to quit>>");
         status = im_receive_input(IM_ALL_SELECT,
                                   IM_INFINITE_TIMEOUT,
                                   &source,
                                   inString);
         printf("\nStatus>%4x",status);
         if (source == IM_KEYBOARD_SELECT) {
             printf("\nFrom Keyboard");
         }
         else if (source == IM_LABEL_SELECT) {
             printf("\nFrom Label");
             im_get_label_symbology(&symbology);
             if (symbology == IM_CODE_39) {
                printf("\nCode 39");
             }
             else {
                printf("\nWrong code!");
             }
         }
         else if (source == IM_COM1_SELECT) {
             printf("\nFrom Serial");
         }
         else {
             printf("\nUnknown source");   // Source = Unknown if reboot
                                            // command or abort command
                                            // received
             valid_source = 0;
         }
         if (valid_source) {
             length = im_get_length(source);
             printf("\nRecvd>>%s",inString);
```

```
            printf("\nLength>%d",length);
            if (inString[0] == 'Q' || inString[0] == 'q')
                done = 1;
        }
        else {
            reboot_flag = im_get_reboot_flag();
            if (reboot_flag) {
                printf("\nReboot flag is set");
                status = im_ready_for_reboot(IM_TRUE);
            }
            else {
                printf("\nReboot flag not set");
                status = im_ready_for_reboot(IM_FALSE);
            }
        }
    }
    im_unlink_comm(IM_COM1);
    im_set_display_mode(IM_SIZE_MODE_80X25,
                        IM_STD_VIDEO_MODE_3,
                        IM_LCD_SCROLL_AT_16,
                        IM_STANDARD_CHAR_HEIGHT);
    im_set_input_mode(IM_WEDGE);
}
```

im_get_viewport_lock

Purpose: This function retrieves the current setting of the viewport lock. When the display is in 80 x 25 mode, you can use the viewport in virtual display mode unless the currently running application program restricts it. The application program may require the viewport to be locked or unlocked.

The viewport lock only affects the user's ability to move the viewport with the viewport control keys.

Syntax:

```
#include "im20lib.h"
IM_STATUS im_get_viewport_lock
    (IM_CONTROL *viewport_lock);
```

IN Parameters: None.

OUT Parameters: The *viewport_lock* parameter is one of these constants:

IM_ENABLE	Viewport is locked
IM_DISABLE	Viewport is unlocked

Return Value: This function returns one of the standard status codes defined in Appendix A, "Status Codes."

Notes: The display must be in 80 x 25 mode (see im_set_display_mode) to use this function.

This function has no effect on the JANUS 2050.

When the viewport is "locked," the viewport movement keys do not move the viewport. The viewport can still move if follow-the-cursor mode is enabled.

See Also: im_set_viewport_lock, im_set_display_mode, im_get_display_mode, im_get_follow_cursor, im_set_follow_cursor

Example

```
/******************* im_get_viewport_lock ******************************/
/* Also with: im_set_viewport_lock  */

#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <stdlib.h>
#include "im20lib.h"

#define  ESC_CHAR      0x1B

void main (void)
{
IM_UCHAR    ch,
            user_option;
IM_CONTROL  current_viewport_lock;

   clrscr();                  /* Clear screen */

   /* Get the current status of viewport lock for restore later */
   im_get_viewport_lock(&current_viewport_lock);

   /* Get user's input option, 'y' for disable viewport lock */
   printf("\nEnable viewport\nlock(y)?");
   user_option = toupper( getch());

   if (user_option == 'Y')
   {
      /* Enable viewport lock */
      im_set_viewport_lock(IM_ENABLE);
      printf("\n\n'ESC' to quit!\nType _f and arrow keys\n");
      printf("See viewport lock!");
   }
   else
   {
      /* Disable viewport lock */
      im_set_viewport_lock(IM_DISABLE);
      printf("\n'ESC' to quit\nType _f and arrow keys\n");
      printf("See viewport unlock!");
   }

   /* Enter characters loop to check viewport movement until 'ESC' key. */
   while ( (ch = getch()) != ESC_CHAR )
      putch(ch);

   /* Restore viewport lock */
   im_set_viewport_lock(current_viewport_lock);
}
```

im_get_warm_boot

Purpose: This function retrieves the current setting of the warm boot status flag. You can enable or disable the **Ctrl-Alt-Del** warm boot key sequence. When disabled, pressing the **Ctrl-Alt-Del** sequence does not reboot the reader.

Syntax:

```
#include "im20lib.h"
IM_STATUS im_get_warm_boot
    (IM_CONTROL *warm_boot);
```

IN Parameters: None.

OUT Parameters: The *warm_boot* parameter is one of these constants:

IM_ENABLE	**Ctrl-Alt-Del** warm boot is enabled
IM_DISABLE	**Ctrl-Alt-Del** warm boot is disabled

Return Value: This function returns one of the standard status codes defined in Appendix A, “Status Codes.”

See Also: im_set_warm_boot

Example

```
/******************* im_get_warm_boot ******************************/
/* Also with: im_set_warm_boot  */

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <ctype.h>
#include "im20lib.h"

void main (void)
{
IM_UCHAR    user_option;
IM_CONTROL  old_warmboot;
IM_CONTROL  new_warmboot;

   clrscr();                  /* Clear screen */
   gotoxy(1,1);               /* Position message */

   /* Get the current status of warn boot */
   im_get_warm_boot(&old_warmboot);

   /* Display warm boot's status */
   if (old_warmboot == IM_ENABLE)
      printf("Warm boot enabled!");
   else
      printf("Warn boot disabled!");

   /* Get user's input option, 'y' for enabling warm boot */
   printf("\n\nEnable warmboot(y)?");
   user_option =  getche();
   user_option = toupper(user_option);

   if (user_option == 'Y')
   {
      /* Warm boot enable */
      im_set_warm_boot(IM_ENABLE);
   }
   else
   {
      /* Warm boot  disable */
      im_set_warm_boot(IM_DISABLE);
   }

   /* Get  new warm boot's status after setting */
   im_get_warm_boot(&new_warmboot);

   /* Display user's input setting */
   if (new_warmboot == IM_ENABLE)
      printf("\nwarm boot enabled!");
   else
      printf("\nWarm boot disabled!");

   printf("\nTest it>>\n");
   getch();
}
```

im_increase_contrast

Purpose: This function increases the LCD display contrast by one level. The display contrast is the lightness or darkness of the characters against the reader display. The reader display has 32 levels of contrast, where 0 is very light, and 31 is very dark.

Syntax:

```
#include <im20lib.h>
IM_STATUS im_increase_contrast(void);
```

IN Parameters: None.

OUT Parameters: None.

Return Value: This function returns one of the standard status codes defined in Appendix A, "Status Codes."

Notes: The 0 to 31 scale fine-tunes the contrast more precisely than using the keypad to adjust the contrast.

The functions im_set_contrast and im_get_contrast use a scale of 0 to 7 that is related to the scale of 0 to 31 used by im_increase_contrast and im_decrease_contrast. The following table compares the two scales.

This Value on Scale 0 to 7	Equates to This Value on Scale 0 to 31	Contrast Level
0	10	Very light
1	12	
2	14	
3	16	
4	18	
5	20	
6	22	
7	24	Very dark

This function has no effect on the JANUS 2050.

See Also: im_get_contrast, im_decrease_contrast, im_set_contrast

Example

```
/******************* im_increase_contrast ******************************/

#include <stdio.h>
#include "im20lib.h"
#include "immsg.h"

void main (void)
{
IM_STATUS status;
status = im_increase_contrast();
   /* If the most significant bit is set, there is an error */
   if (IM_ISERROR(status))
      printf("Inc Contrast error = %x\n", status);
}
```

im_input_status

Purpose: This function checks to see if any input buffers have data and returns the buffer identification.

Syntax:
```
#include <im20lib.h>
IM_ORIGIN im_input_status (void);
```

IN Parameters: None.

OUT Parameters: None.

Return Value: IM_ORIGIN is one or more of these constants:

IM_NO_SELECT	No input buffer selected
IM_LABEL_SELECT	Label selected
IM_KEYBOARD_SELECT	Keypad selected
IM_COM1_SELECT	COM1 selected
IM_COM2_SELECT	COM2, scanner port selected (2010 and 2050)
IM_COM4_SELECT	COM4 selected (RF only)
IM_ALL_SELECT	All input buffers are selected

Notes: You must install the Reader Wedge to use this function. For information on loading RWTSR.EXE, see “Runtime Requirements” earlier in this chapter.

See Also: im_receive_input

Example

```
/******************* im_input_status  *****************************/
/* Also with: im_receive_input  */
/* im_get_length   */

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include "im20lib.h"
#include "immsg.h"

void main (void)
{
IM_UCHAR   input[300];
IM_MODE    original_mode;
IM_ORIGIN  source;
IM_STATUS  status;

   window(1, 1, 20, 16);       /* Set the application text window */
   clrscr();                   /* Clear the screen  */

   printf("Demo im_input_status\n");

   /* Get the previous input mode for restoring input mode later */
   original_mode = im_get_input_mode();

   /* Set Reader Wedge mode to program interface */
   im_set_input_mode(IM_DESKTOP);

   /* Loop until some input */
   while ( (source = im_input_status()) == IM_NO_SELECT)
      ;

   /* See where the input comes from */
   if ( source == IM_COM1_SELECT)
      printf("COM1 input:");
   else
      printf("COM4 input:");

   /* Request input from source */
   status = im_receive_input(source, IM_INFINITE_TIMEOUT, &source, input);

   if ( IM_ISGOOD(status) )
      printf("\n%s\n", input);      /* Display input data */
   else   /* input error */
   {
      printf("\n");
      im_message(status);
   }

   /* Restore original input mode */
   im_set_input_mode(original_mode);
}
```

im_irl_a

Purpose: This function returns input from bar code labels or the keypad in the same manner as IRL ASCII input command A. This function returns the input data to the buffer, edits reader commands, and displays input data. For more information on IRL and command A, refer to the *IRL Programming Reference Manual*.

Syntax:

```
#include <im20lib.h>
IM_STATUS im_irl_a
    (IM_USHORT timeout,
    IM_LENGTH_SPEC test_table[],
    IM_UCHAR mask_string[],
    IM_UCHAR *instring,
    IM_USHORT *cmd_count,
    IM_DECTYPE *symbology);
```

IN Parameters: The *timeout* parameter is the receive timeout period. You can exit the function before data is received or before the timeout occurs by performing an application break sequence (see your JANUS user's manual). The return status indicates whether the function was successful, a timeout occurred, or the application break was received.

The *timeout* parameter is a number or one of these constants:

1 to 65,534 ms	Numeric range
IM_ZERO_TIMEOUT	No wait
IM_INFINITE_TIMEOUT	Wait forever

If IM_INFINITE_TIMEOUT is selected, the function will not return until the end of message character has been received.

Data is returned only if its length matches one of the five lengths specified in the *test_table.* The *test_table* parameter must be a matrix of the following form:

{a, b, c, d},
{a, b, c, d},
{a, b, c, d},
{a, b, c, d},
{a, b, c, d},

The *a* position in the matrix is one of the following:

IM_NO_LENGTH	Accept data of any length
IM_LENGTH	Accept data with a specific length
IM_RANGE	Accept data within a length range

If IM_LENGTH is specified in the *a* position, the actual length of the data string is placed in the *d* position (and *b* and *c* are not used).

If IM_RANGE is specified in the *a* position, the data length must be within the range of *b* and *c* (and *d* is not used).

Set any unused table entries to {IM_NO_LENGTH,0,0,0}.

The *mask_string* parameter sets up a data mask that received data must match. *mask_string* can accept a string of constants or wildcard characters. For example, use the string ### - #### to accept only phone numbers.

You can use one or more of these wildcard characters to define the mask:

#	Numeric
@	Alpha
?	Alphanumeric printable
NULL (CHR$(0))	No mask

OUT Parameters: The *instring* parameter is the input string. You must allocate at least 256 bytes for *instring*.

The *cmd_count* returned is 0 unless an abort command is passed in the string.

The *symbology* parameter is one of these constants:

IM_UNKNOWN_DECODE	Unknown bar code
IM_CODABAR	Codabar bar code
IM_CODE_11	Code 11 bar code
IM_CODE_16K	Code 16K bar code
IM_CODE_39	Code 39 bar code
IM_CODE_49	Code 93 bar code
IM_CODE_93	Code 49 bar code
IM_CODE_128	Code 128 bar code
IM_I_2_OF_5	Interleaved 2 of 5
IM_MSI	MSI bar code
IM_PDF_417	PDF 417 bar code
IM_PLESSEY	Plessey bar code
IM_UPC	Universal Product code

Return Value: This function returns one of the standard status codes defined in Appendix A, "Status Codes."

Notes: You must set the reader to Programmer mode with im_set_input_mode.

You must install the Reader Wedge to use this function. For information on loading RWTSR.EXE, see "Runtime Requirements" earlier in this chapter.

Data does not display correctly on the screen if the display is in the 80 x 25 mode after the cursor has scrolled off the bottom of the display. Set the display to one of the other modes using im_set_display_mode.

See Also: im_irl_k, im_irl_n, im_irl_v, im_irl_y, im_set_display_mode, im_set_input_mode

Example

```
/******************* im_irl_a ************************************/
/* Also with: im_set_input_mode */
/*   im_standby_wait            */

#include <stdio.h>
#include <conio.h>
#include "im20lib.h"

IM_LENGTH_SPEC length_table[IM_LENGTH_SPEC_MAX] = {
    {IM_LENGTH,   0,   0,   2},
    {IM_RANGE,    7,  10,   0},
    {IM_LENGTH,   0,   0,   4},
    {IM_RANGE,   15,  17,   0},
    {IM_LENGTH,   0,   0,   6}
};

IM_UCHAR ssn_mask[] = "###-##-####";

void main (void)
{
IM_UCHAR   input[256];
IM_USHORT  cc;
IM_DECTYPE symbol;

   window(1, 1, 20, 16);       /* Set the application text window */
   clrscr();                   /* Clear the screen  */

   printf("Demo IRL A Bar Code\n");
   printf(" '9' = 1st char to quit\n");
   printf(" Example: '933-11-3333' \n");

   /* Set Reader Wedge into PROGRAMMER mode  */
   im_set_input_mode(IM_PROGRAMMER);

   /* Input loop  */
   do
   {
      /* Test wait for 5 seconds */
      printf("Start waiting for 5 seconds\n");
      im_standby_wait(5000);
      printf("Done waiting for 5 seconds\n");

      /* Display IRL A mask pattern */
      printf("IRL A test mask\n");
      printf(" = %s\n", ssn_mask);

      /* Request input from Reader Wedge */
      im_irl_a(IM_INFINITE_TIMEOUT, length_table,
               ssn_mask, input, &cc, &symbol);

      printf("\n%s\n", input);

   } while (input[0] != '9');  /* '9' to quit */

   /* Switch the Reader Wedge back to virtual wedge mode */
   im_set_input_mode(IM_WEDGE);
}
```

im_irl_k

Purpose: This function receives input from the keypad in any format in the same manner as IRL ASCII input command K. This function returns the input data to the buffer, edits reader commands, and displays input data. For more information on IRL and command K, refer to the *IRL Programming Reference Manual*.

Syntax:

```
#include <im20lib.h>
IM_STATUS im_irl_k
    (IM_USHORT timeout,
    IM_LENGTH_SPEC test_table[],
    IM_UCHAR mask_string[],
    IM_UCHAR *instring,
    IM_USHORT *cmd_count);
```

IN Parameters: The *timeout* parameter is the receive timeout period. You can exit the function before data is received or before the timeout occurs by performing an application break sequence (see your JANUS user's manual). The return status indicates whether the function was successful, a timeout occurred, or the application break was received.

The *timeout* parameter is a number or one of these constants:

1 to 65,534 ms	Numeric range
IM_ZERO_TIMEOUT	No wait
IM_INFINITE_TIMEOUT	Wait forever

If IM_INFINITE_TIMEOUT is selected, the function will not return until the end of message character has been received.

Data is returned only if its length matches one of the five lengths specified in the *test_table*. The *test_table* parameter must be a matrix of the following form:

{a, b, c, d},
{a, b, c, d},
{a, b, c, d},
{a, b, c, d},
{a, b, c, d},

The *a* position in the matrix is one of the following:

IM_NO_LENGTH	Accept data of any length
IM_LENGTH	Accept data of a specific length
IM_RANGE	Accept data within a length range

If IM_LENGTH is specified in the *a* position, the actual length of the data string is placed in the *d* position (and *b* and *c* are not used).

If IM_RANGE is specified in the *a* position, the data length must be within the range of *b* and *c* (and *d* is not used).

Set any unused table entries to {IM_NO_LENGTH,0,0,0}.

The *mask_string* parameter sets up a data mask that received data must match. *mask_string* can accept a string of constants or wildcard characters. For example, use the string ### - #### to accept only phone numbers.

You can use one or more of these wildcard characters to define the mask:

#	Numeric
@	Alpha
?	Alphanumeric printable
NULL (CHR$(0))	No mask

OUT Parameters: The *instring* parameter is the input string. You must allocate at least 256 bytes for *instring*.

The *cmd_count* returned is 0 unless an abort command is passed in the string.

Return Value: This function returns one of the standard status codes defined in Appendix A, “Status Codes.”

Notes: You must set the reader to Programmer mode with im_set_input_mode.

You must install the Reader Wedge to use this function. For information on loading RWTSR.EXE, see “Runtime Requirements” earlier in this chapter.

Data does not display correctly on the screen if the display is in the 80 x 25 mode after the cursor has scrolled off the bottom of the display. Set the display to one of the other modes using im_set_display_mode.

See Also: im_irl_a, im_irl_n, im_irl_v, im_irl_y, im_set_display_mode, im_set_input_mode

Example

```
/******************* im_irl_k ************************************/
/* Also with: im_set_input_mode  and  im_standby_wait               */

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <ctype.h>
#include "im20lib.h"

IM_LENGTH_SPEC length_table[IM_LENGTH_SPEC_MAX] = {
   {IM_LENGTH,   0,   0,   2},
   {IM_RANGE,    7,  10,   0},
   {IM_LENGTH,   0,   0,   4},
   {IM_RANGE,   15,  17,   0},
   {IM_LENGTH,   0,   0,   6}
};

IM_UCHAR mix_mask[] = "?#@#";

void main (void)
{
IM_UCHAR   input[256];
IM_USHORT  cc;

   window(1, 1, 20, 16);       /* Set the reader services text window */
   clrscr();                   /* Clear the screen  */
   printf("Demo IRL K Bar Code\n");
   printf(" 'Q' = 1st char to quit\n");
   printf("Example: ‘Q1q2’ \n");

   /* Set Reader Wedge into PROGRAMMER mode */
   im_set_input_mode(IM_PROGRAMMER);

   /* Input loop --Display IRL K mask pattern */
   do
   {
      printf("IRL K test mask:\n");
      printf( "   = %s \n", mix_mask);
      printf("Example: ‘A5a3’\n\n");

      /* Request input from Reader Wedge */
      im_irl_k(IM_INFINITE_TIMEOUT, length_table,
               mix_mask, input, &cc);

      printf("\n%s\n", input);

      /* Upper case first char of input for testing quit */
      input[0] = toupper(input[0]);
   } while (input[0] != 'Q');              /* 'Q' to quit */

   /* Switch the Reader Wedge back to virtual wedge mode */
   im_set_input_mode(IM_WEDGE);
}
```

im_irl_n

Purpose: This function receives numeric input from the keypad or a label in the same manner as IRL numeric input command N. Nonnumeric data is ignored.

This function clears any data existing in the keypad or label buffers before the function was called, edits reader commands from input, and displays input data. For more information on IRL and command N, refer to the *IRL Programming Reference Manual.*

Syntax:

```
#include <im20lib.h>
IM_STATUS im_irl_n
    (IM_USHORT timeout,
    IM_LENGTH_SPEC test_table[],
    IM_UCHAR *instring,
    IM_USHORT *cmd_count,
    IM_DECTYPE *symbology);
```

IN Parameters: The *timeout* parameter is the receive timeout period. You can exit the function before data is received or before the timeout occurs by performing an application break sequence (see your JANUS user's manual). The return status indicates whether the function was successful, a timeout occurred, or the application break was received.

The *timeout* parameter is a number or one of these constants:

1 to 65,534 ms	Numeric range
IM_ZERO_TIMEOUT	No wait
IM_INFINITE_TIMEOUT	Wait forever

If IM_INFINITE_TIMEOUT is selected, the function will not return until the end of message character has been received.

Data is returned only if its length matches one of the five lengths specified in the *test_table.* The *test_table* parameter must be a matrix of the following form:

{a, b, c, d},
{a, b, c, d},
{a, b, c, d},
{a, b, c, d},
{a, b, c, d},

The *a* position in the matrix is one of the following:

IM_NO_LENGTH	Accept data of any length
IM_LENGTH	Accept data of a specific length
IM_RANGE	Accept data within a length range

If IM_LENGTH is specified in the *a* position, the actual length of the data string is placed in the *d* position (and *b* and *c* are not used).

If IM_RANGE is specified in the *a* position, the data length must be within the range of *b* and *c* (and *d* is not used).

Set any unused table entries to {IM_NO_LENGTH,0,0,0}.

OUT Parameters: The *instring* parameter is the input string. You must allocate at least 256 bytes for *instring*.

The *cmd_count* returned is 0 unless an abort command is passed in the string.

The *symbology* parameter is one of these constants:

IM_UNKNOWN_DECODE	Unknown bar code
IM_CODABAR	Codabar bar code
IM_CODE_11	Code 11 bar code
IM_CODE_16K	Code 16K bar code
IM_CODE_39	Code 39 bar code
IM_CODE_49	Code 93 bar code
IM_CODE_93	Code 49 bar code
IM_CODE_128	Code 128 bar code
IM_I_2_OF_5	Interleaved 2 of 5
IM_MSI	MSI bar code
IM_PDF_417	PDF 417 bar code
IM_PLESSEY	Plessey bar code
IM_UPC	Universal Product code

Return Value: This function returns one of the standard status codes defined in Appendix A, “Status Codes.”

Notes: You must set the reader to Programmer mode with im_set_input_mode.

You must install the Reader Wedge to use this function. For information on loading RWTSR.EXE, see "Runtime Requirements" earlier in this chapter.

Data does not display correctly on the screen if the display is in the 80 x 25 mode after the cursor has scrolled off the bottom of the display. Set the display to one of the other modes using im_set_display_mode.

See Also: im_irl_a, im_irl_k, im_irl_v, im_irl_y, im_set_input_mode

Example

```
/******************* im_irl_n ************************************/
#include <stdio.h>
#include <conio.h>
#include "im20lib.h"

IM_LENGTH_SPEC length_table[IM_LENGTH_SPEC_MAX] = {
   {IM_LENGTH,   0,   0,   2},
   {IM_RANGE,    7,  10,   0},
   {IM_LENGTH,   0,   0,   4},
   {IM_RANGE,   15,  17,   0},
   {IM_LENGTH,   0,   0,   6}
};

void main (void)
{
IM_UCHAR   input[256];
IM_USHORT  cc;
IM_DECTYPE symbol;

   window(1, 1, 20, 16);       /* Set the reader services text window */
   clrscr();                   /* Clear the screen  */
   printf("Demo IRL N Bar Code\n");
   printf(" '9' to quit\n");

   /* Set Reader Wedge mode to program interface */
   im_set_input_mode(IM_PROGRAMMER);

   /* Input loop */
   do
   {

      /* Display IRL N test */
      printf("IRL N test\n");

      /* Request input from Reader Wedge */
      im_irl_n(IM_INFINITE_TIMEOUT, length_table,
                input, &cc, &symbol);

      printf("\n%s\n",input);

   } while (input[0] != '9' );         /* '9' to quit */

   /* Switch the Reader Wedge back to virtual wedge mode  */
   im_set_input_mode(IM_WEDGE);
}
```

im_irl_v

Purpose: This function receives input from any specified source in any format, in the same manner as an IRL universal input command V. For more information on IRL and command V, refer to the *IRL Programming Reference Manual.*

Syntax:

```
#include <im20lib.h>
IM_STATUS im_irl_v
    (IM_USHORT timeout,
    IM_CONTROL edit,
    IM_LABEL_BEEP_CONTROL beep,
    IM_CONTROL display,
    IM_ORIGIN *source,
    IM_UCHAR *instring,
    IM_USHORT *cmd_count,
    IM_DECTYPE *symbology);
```

IN Parameters: The *timeout* parameter is the receive timeout period. You can exit the function before data is received or before the timeout occurs by performing an application break sequence (see your JANUS user's manual). The return status indicates whether the function was successful, a timeout occurred, or the application break was received.

The *timeout* parameter is a number or one of these constants:

1 to 65,534 ms	Numeric range
IM_ZERO_TIMEOUT	No wait
IM_INFINITE_TIMEOUT	Wait forever

If IM_INFINITE_TIMEOUT is selected, the function will not return until the end of message character has been received.

The *edit* parameter determines whether the Reader Wedge parses reader commands. Use one of these constants:

IM_DISABLE	Disable reader command parsing
IM_ENABLE	Enable reader command parsing

If *edit* is IM_DISABLED, reader commands are treated as data.

The *beep* parameter determines whether the IRL V command beeps or not when data is entered.

If *beep* is set to IM_APPLI_BEEP, the Reader Wedge does not control the beep. Instead, you code the application to sound a beep when your program design requires one.

If beep is set to IM_WEDGE_BEEP, the reader always beeps when data is entered. The *beep* parameter is one of these constants:

IM_APPLI_BEEP	Application controls the beep
IM_WEDGE_BEEP	Beeps occur automatically

The *display* parameter determines if the data is displayed as it is entered. The *display* parameter is one of these constants:

IM_DISABLE	Disable display of data
IM_ENABLE	Enable display of data

IN/OUT Parameters:

The *source* parameter determines which input sources are allowed. When the IRL V command returns, *source* indicates where the data came from. When both keypad and label inputs are allowed, the *source* always returns keypad because it is possible for keyed and scanned data to be intermixed. Although intermixing is possible, it is not likely to occur.

If you have both the keypad and scanner enabled, and you want to know where the data came from, first check the symbology:

- If the symbology is IM_UNKNOWN_DECODE, then the data came from the keypad.
- If the symbology is one of the other values (such as IM_CODE_39), then some or all of the data came from the scanner.
- If only the scanner is enabled, then all of the data came from the scanner.

The *source* parameter is one of these constants:

IM_NO_SELECT	No source selected
IM_LABEL_SELECT	Label selected
IM_KEYBOARD_SELECT	Keypad selected
IM_COM1_SELECT	COM1 selected
IM_COM2_SELECT	COM2, scanner port selected (2010 and 2050)
IM_COM4_SELECT	COM4 selected (RF only)
IM_ALL_SELECT	All sources are selected

OUT Parameters: The *instring* parameter is the input string. You must allocate at least 256 bytes for *instring*.

The *cmd_count* returned is 0 unless an abort command is passed in the string.

The *symbology* parameter is one of these constants:

IM_UNKNOWN_DECODE	Unknown bar code
IM_CODABAR	Codabar bar code
IM_CODE_11	Code 11 bar code
IM_CODE_16K	Code 16K bar code
IM_CODE_39	Code 39 bar code
IM_CODE_49	Code 93 bar code
IM_CODE_93	Code 49 bar code
IM_CODE_128	Code 128 bar code
IM_I_2_OF_5	Interleaved 2 of 5
IM_MSI	MSI bar code
IM_PDF_417	PDF 417 bar code
IM_PLESSEY	Plessey bar code
IM_UPC	Universal Product code

Return Value: This function returns one of the standard status codes defined in Appendix A, “Status Codes.”

Notes: You must set the reader to Programmer mode with im_set_input_mode.

To receive data from a communications port, you must install a protocol handler.

You must install the Reader Wedge to use this function. For information on loading RWTSR.EXE, see “Runtime Requirements” earlier in this chapter.

Data is automatically cleared from both the scanner and the keypad buffers but not from any communications port buffer.

Use the im_link_comm function to establish a link between the Reader Wedge function and a communications port, and use the im_unlink_comm function to remove the link.

Data does not display correctly on the screen if the display is in the 80 x 25 mode after the cursor has scrolled off the bottom of the display. Set the display to one of the other modes using im_set_display_mode.

See Also: im_irl_a, im_irl_k, im_irl_n, im_irl_y, im_set_input_mode

Example

```
/******************* im_irl_v ************************************/
/* Also with: im_link_comm   im_unlink_comm   im_set_input_mode      */
/* Note: Load PHIMEC.EXE protocol handler before using this program. */

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <ctype.h>
#include "immsg.h"
#include "im20lib.h"

#define  COM_BUFSIZE 300        /* allocate 300 bytes for comm buffer */

void main (void)
{
IM_UCHAR        *com_buffer;
IM_UCHAR        input[256];
IM_ORIGIN       source;
IM_USHORT       cc;
IM_DECTYPE      symbol;

   /* FUNCTION BODY */

   window(1, 1, 20, 16);       /* Set the reader services text window */
   clrscr();                   /* Clear the screen  */
   printf("Demo IRL V Bar code\n'C' to clear screen\n'Q' to quit\n");

/* Allocate a comm buffer */
   com_buffer = (char *) malloc(COM_BUFSIZE);
/* Set Reader Wedge mode to program interface */
   im_set_input_mode(IM_PROGRAMMER);
/* Link com_buffer to com1 */
   im_link_comm(IM_COM1, com_buffer, COM_BUFSIZE);
/* im_irl_v input loop */
   do
   {
/* Display IRL V test */
      printf("IRL V test\n");
/* Set up input source with labels, keypad, and com1 */
      source = IM_LABEL_SELECT | IM_KEYBOARD_SELECT | IM_COM1_SELECT;
/* Request input from Reader Wedge */
      im_irl_v(IM_INFINITE_TIMEOUT,
               IM_ENABLE, IM_WEDGE_BEEP, IM_ENABLE,
               &source, input, &cc, &symbol);
/* Display input data */
      printf("\n%s\n", input);
/* Upper case first char of input for simplifying to test input */
      input[0] = toupper(input[0]);
/* If the first char in string is 'C', then clear screen.*/
      if (input[0] == 'C')
         clrscr();
} while (input[0] != 'Q');        /* 'Q' for quit */
/* Unlink from comm port */
   im_unlink_comm(IM_COM1);
/* Free allocated memory */
   free(com_buffer);
/* Switch the Reader Wedge back to virtual wedge mode  */
   im_set_input_mode(IM_WEDGE);
im_set_beep_control(IM_WEDGE_BEEP);
}
```

im_irl_y

Purpose: The im_irl_y function receives input from the designated communications port the same as IRL ASCII input command Y. This function receives a single block, not an entire file. This function always clears input from the host and edits reader commands from input. Unlike the other IRL instructions, the data is not automatically displayed. For more information on IRL and command Y, refer to the *IRL Programming Reference Manual.*

Syntax:

```
#include <im20lib.h>
IM_STATUS im_irl_y
    (IM_USHORT timeout,
    IM_COM_PORT port_id,
    IM_UCHAR *eom_char,
    IM_PROTOCOL_CMD protocol,
    IM_UCHAR *instring,
    IM_USHORT *cmd_count);
```

IN Parameters: The *timeout* parameter is the receive timeout period. You can exit the function before data is received or before the timeout occurs by performing an application break sequence (see your JANUS user's manual). The return status indicates whether the function was successful, a timeout occurred, or the application break was received.

The *timeout* parameter is a number or one of these constants:

1 to 65,534 ms	Numeric range
IM_ZERO_TIMEOUT	No wait
IM_INFINITE_TIMEOUT	Wait forever

If IM_INFINITE_TIMEOUT is selected, the function will not return until the end of message character has been received.

The *port_id* parameter identifies the communications port as follows:

IM_COM1	COM1
IM_COM2	COM2, scanner port (2010 and 2050)
IM_COM4	COM4 (RF only)

The *eom_char* parameter is the end of message character needed to terminate input from the communications port. The character is normally a carriage return, but you can set it to any other character.

The *protocol* parameter is one of these constants:

IM_NO_CHANGE	Keep the current protocol
IM_PROTOCOL_OFF	Turn the protocol OFF
IM_PROTOCOL_ON	Turn the protocol ON
IM_NEW_TERM_CHAR	Use the new termination character

The *protocol* parameter affects how incoming messages are received:

- If IM_NO_CHANGE is specified, the protocol and end of message settings remain the same as the last time the function was called.
- If IM_PROTOCOL_OFF is specified, the incoming data is terminated with the specified end of message character. If the end of message character is null, the function terminates upon receiving the first character.
- If IM_PROTOCOL_ON is specified, the incoming message is terminated with the end of message character of the active protocol.
- If IM_NEW_TERM_CHAR is specified, IM_PROTOCOL_OFF is assumed and the new end of message character specified in the parameter list is used.

OUT Parameters: The *instring* parameter is the input string. You must allocate at least 256 bytes for *instring*.

The *cmd_count* returned is 0 unless an abort command is passed in the string.

Return Value: This function returns one of the standard status codes defined in Appendix A, “Status Codes.”

Notes: The receive buffers are cleared whenever the protocol is turned ON or OFF. To change the termination character (or to not change the setting at all), use IM_NEW_TERM_CHAR or IM_NO_CHANGE instead of toggling the protocol.

Data does not display correctly on the screen if the display is in the 80 x 25 mode after the cursor has scrolled off the bottom of the display. Set the display to one of the other modes using im_set_display_mode.

Set the reader to Programmer mode using im_set_input_mode to use this function.

You must install a protocol handler to use this function.

You must install the Reader Wedge to use this function. For information on loading RWTSR.EXE, see "Runtime Requirements" earlier in this chapter.

See Also: im_irl_a, im_irl_k, im_irl_n, im_irl_v, im_set_display_mode, im_set_input_mode

Example

```
/******************** im_irl_y ************************************/
/* Also with: im_link_comm */
/*   im_unlink_comm        */
/*   im_set_input_mode     */

/* Note: Load JANUS's PHIMEC protocol before using this sample program. */

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include "immsg.h"
#include "im20lib.h"

#define COM_BUFSIZE 300      /* Allocate 300 bytes for comm buffer */

static IM_UCHAR   eom_ctrl_a = 1;       /* Control-A for EOM char       */
static IM_UCHAR   eom_ctrl_y = 25;      /* Control-Y for EOM char       */
static IM_UCHAR   eom_ctrl_z = 26;      /* Control-Z for EOM char       */
static IM_UCHAR   eom_cr     = '\r';    /* Carriage Return for EOM char */
static IM_UCHAR   eom_null   = '\0';    /* Null char for EOM char       */

void main (void)
{
IM_UCHAR        *com_buffer;
IM_UCHAR        input[256];
IM_USHORT       cc;
IM_STATUS status;

   window(1, 1, 20, 16);       /* Set the application text window */
   clrscr();                   /* Clear the screen  */

   printf("Demo IRL Y Bar Code\n'Q' to quit\n\'C' to clear screen\n");

   /* Allocate a comm buffer */
   com_buffer = (char *) malloc(COM_BUFSIZE);
```

```
/* Link com_buffer to com1 */
im_link_comm(IM_COM1, com_buffer, COM_BUFSIZE);

/* Set Reader Wedge mode to program interface */
im_set_input_mode(IM_PROGRAMMER);

/* Using JANUS's protocol by passing IM_PROTOCOL_ON */
printf("\nUsing JANUS's protocol\n");
status = im_irl_y(IM_INFINITE_TIMEOUT, IM_COM1, &eom_null,
                          IM_PROTOCOL_ON, input, &cc);
/* Display input data */
printf("\nstatus: %x\n%s", status, input);

/*********************************************************************/
/* The following im_irl_y() calls demonstrates that the various types  */
/* of EOM chars are used for EOM purpose. If you just use one type EOM */
/* char, then you delete the unwanted im_irl_y() call.                 */
/*********************************************************************/

/* Not using JANUS's protocol by passing IM_PROTOCOL_OFF and new EOM   */
/* char control-y                                                      */
printf("\nCTRL-Y for EOM\n");
status = im_irl_y(IM_INFINITE_TIMEOUT, IM_COM1, &eom_ctrl_y,
                          IM_PROTOCOL_OFF, input, &cc);
/* Display input data */
printf("\nstatus: %x\n%s", status, input);

/* Using IM_NEW_TERM_CHAR and changing EOM char to control-z */
printf("\nCTRL-Z for EOM\n");
status = im_irl_y(IM_INFINITE_TIMEOUT, IM_COM1, &eom_ctrl_z,
                          IM_NEW_TERM_CHAR, input, &cc);

/* Display input data */
printf("\nstatus: %x\n%s", status, input);

/* Using IM_PROTOCOL_OFF and changing EOM char to control-a */
printf("\nCTRL-A for EOM\n");
status = im_irl_y(IM_INFINITE_TIMEOUT, IM_COM1, &eom_ctrl_a,
                          IM_PROTOCOL_OFF, input, &cc);

/* Display input data */
printf("\nstatus: %x\n%s", status, input);

/* Using JANUS's protocol for input again */
printf("\nUsing JANUS's protocol\n");
status = im_irl_y(IM_INFINITE_TIMEOUT, IM_COM1, &eom_null,
                          IM_PROTOCOL_ON, input, &cc);

/* im_irl_y input loop */
do
{
   /* Using JANUS's protocol, If using Pt-to-Pt protocol, EOM is CR */
   printf("\nCR is EOM of Pt-to-Pt\n");
   status = im_irl_y(IM_INFINITE_TIMEOUT, IM_COM1, &eom_null,
                          IM_NO_CHANGE, input, &cc);

   /* Display input data */
   printf("\nstatus: %x\n%s", status, input);
   /* Test input Y with protocol and timeout */

   /* Upper case first char of input for simplifying to test input */
   input[0] = toupper(input[0]);
```

```
      /*If the first char in string is 'C', then clear screen.*/
      if (input[0] == 'C')
         clrscr();

   } while (input[0] != 'Q');  /* 'Q' to stop */

   /* Unlink com_buffer from com1 */
   im_unlink_comm(IM_COM1);

   /* Free allocated memory */
   free(com_buffer);

   /* Switch the Reader Wedge back to virtual wedge mode */
   im_set_input_mode(IM_WEDGE);
}
```

IM_ISERROR

Purpose: This macro determines if the return status code from another PSK function is an error (either fatal or nonfatal).

Syntax:
```
#include <im20lib.h>
IM_ISERROR(status);
```

IN Parameters: The *status* parameter is any PSK function that returns a status code.

OUT Parameters: None.

Return Value:

0	Indicates success or warning
nonzero	Indicates an error (either fatal or nonfatal)

See Also: "Status Code Macros" earlier in this chapter.
IM_ISGOOD, IM_ISSUCCESS, IM_ISWARN

Example

```
/******************* IM_ISERROR ************************************/
#include "im20lib.h"
#include <stdio.h>
printf("IM_ISERROR example:")

status = im_sound(IM_LOW_PITCH, IM_BEEP_DURATION, IM_NORMAL_VOLUME)
if IM_ISERROR(status)
   printf("Beep error!");
else
   printf("Beep success or warning!");
```

IM_ISGOOD

Purpose: This macro determines if the return status code from another PSK function is success.

Syntax:
```
#include <im20lib.h>
IM_ISGOOD(status);
```

IN Parameters: The *status* parameter is any PSK function that returns a status code.

OUT Parameters: None.

Return Value:

0	Indicates a warning or an error (either fatal or nonfatal)
nonzero	Indicates success

See Also: "Status Code Macros" earlier in this chapter.
IM_ISERROR, IM_ISSUCCESS, IM_ISWARN

Example

```
/******************* IM_ISGOOD ************************************/
#include "im20lib.h"
#include <stdio.h>
printf("IM_ISGOOD example:")

status = im_sound(IM_LOW_PITCH, IM_BEEP_DURATION, IM_NORMAL_VOLUME)
if IM_ISGOOD(status)
   printf("Beep successful!");
else
   printf("Beep warning or error!");
```

IM_ISSUCCESS

Purpose: This macro determines if the return status code from another PSK function is either success or warning.

Syntax:
```
#include <im20lib.h>
IM_ISSUCCESS(status);
```

IN Parameters: The *status* parameter is any PSK function that returns a status code.

OUT Parameters: None.

Return Value:

0	Indicates an error (either fatal or nonfatal)
nonzero	Indicates success or warning

See Also: "Status Code Macros" earlier in this chapter.
IM_ISERROR, IM_ISGOOD, IM_ISWARN

Example

```
/******************* IM_ISSUCCESS ************************************/
#include "im20lib.h"
#include <stdio.h>
printf("IM_ISSUCCESS example:")

status = im_sound(IM_LOW_PITCH, IM_BEEP_DURATION, IM_NORMAL_VOLUME)
if IM_ISSUCCESS(status)
   printf("Beep success or warning!");
else
   printf("Beep error!");
```

IM_ISWARN

Purpose: This macro determines if the return status code from another PSK function is a warning.

Syntax:
```
#include <im20lib.h>
IM_ISWARN(status);
```

IN Parameters: The *status* parameter is any PSK function that returns a status code.

OUT Parameters: None.

Return Value:

0	Indicates success or an error (either fatal or nonfatal)
nonzero	Indicates a warning

See Also: "Status Code Macros" earlier in this chapter.
IM_ISERROR, IM_ISGOOD, IM_ISSUCCESS

Example

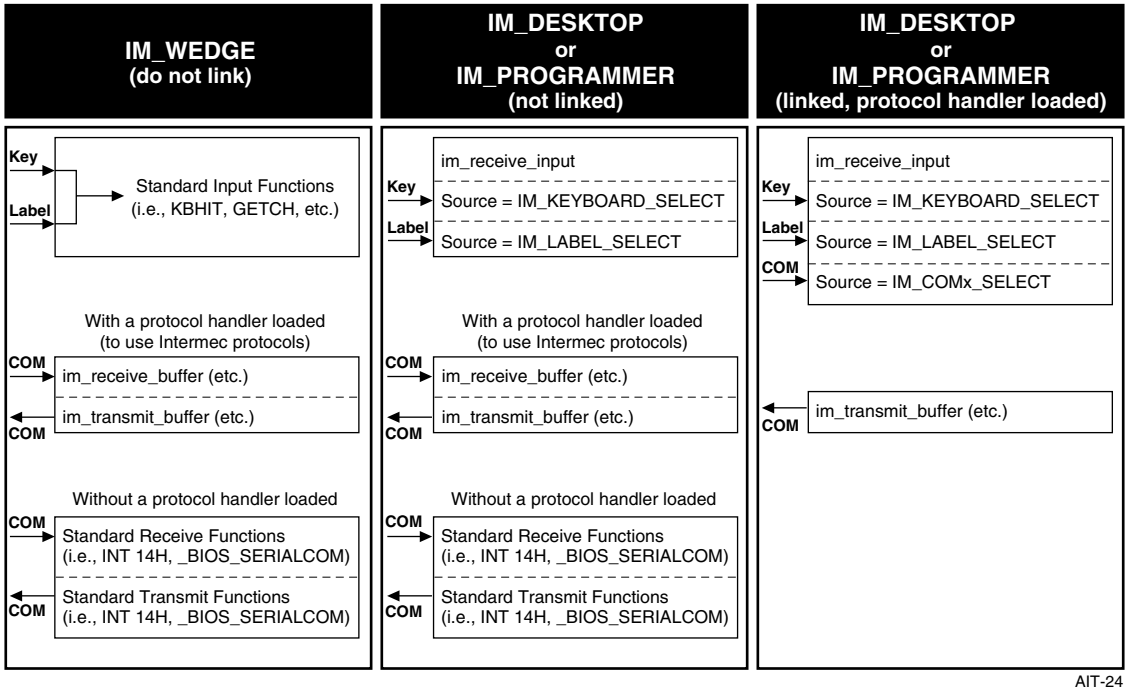
```
/******************* IM_ISWARN ************************************/
#include "im20lib.h"
#include <stdio.h>
printf("IM_ISWARN example:")

status = im_sound(IM_LOW_PITCH, IM_BEEP_DURATION, IM_NORMAL_VOLUME)
if IM_ISWARN(status)
   printf("Beep warning!");
else
   printf("Beep success or error!");
```

im_link_comm

Purpose: This function establishes a link between the Reader Wedge function and a designated communications port. The following figure lists the different modes and the functions to use when linked or unlinked.

Note: *Programmer mode and desktop mode differ in how they handle keypad inputs. When the reader is in Programmer mode, the* ***Ente****r key terminates keypad input. When the reader is in Desktop mode, keypad entry terminates when you press each key.*

Linking Specific Modes and Functions



Syntax:

```
#include <im20lib.h>
IM_STATUS im_link_comm
    (IM_COM_PORT port_id,
    void *buffer,
    IM_USHORT buffer_size);
```

IN Parameters: The *port_id* parameter identifies the communications port as follows:

IM_COM1	COM1
IM_COM2	COM2, scanner port (2010 and 2050)
IM_COM4	COM4 (RF only)

The *buffer* parameter is a pointer to a memory area used by the Reader Wedge. Your application should not use this buffer.

The *buffer_size* parameter is the number of bytes to allocate for the buffer array. The suggested size is 300 bytes.

OUT Parameters: None.

Return Value: This function returns one of the standard status codes defined in Appendix A, "Status Codes."

Notes: Use this function if you want to receive data with the Reader Wedge input function im_irl_y or to use the communications port with im_receive_input and im_irl_v.

This function needs to be called only once.

You must unlink at the end of your application.

You must install a protocol handler to use this function.

You must install the Reader Wedge to use this function. For information on loading RWTSR.EXE, see "Runtime Requirements" earlier in this chapter.

See Also: im_unlink_comm, im_receive_input, im_irl_v, im_irl_y, im_serial_protocol_control

---

Example

```
See example for im_receive_input.
```

fim_message

Purpose: This function displays the error message associated with a specific status code returned by an Intermec function. These status codes are listed in Appendix A.

Use this function to display additional information about status codes during application development.

Syntax:

```
#include "im20lib.h"
void im_message(IM_STATUS status_code);
```

IN Parameters: The status_code parameter is a standard status code returned from various PSK functions.

OUT Parameters: None.

Return Value: None.

Notes: The status message is displayed at the current cursor location without any formatting.

Example

```
See example for im_set_contrast.
```

im_number_pad_off

Purpose: This function disables the number pad feature. When the number pad is disabled, the numeric keys function the same as the **1** through **+** keys above the "QWERTY" keys on a normal PC keyboard.

When the number pad is disabled, you cannot type characters from the extended ASCII character set or use the number pad to move the cursor. For more information, refer to your JANUS user's manual.

Syntax:

```
#include <im20lib.h>
IM_STATUS im_number_pad_off (void);
```

IN Parameters: None.

OUT Parameters: None.

Return Value: This function returns one of the standard status codes defined in Appendix A, "Status Codes."

See Also: im_number_pad_on

Example

```
See example for Im_number_pad_on.
```

im_number_pad_on

Purpose: This function enables the reader's number pad. When enabled, the number pad functions the same way as the 10-key number pad on a standard PC keyboard. The numeric keys then provide cursor movement and editing keys, numbers, and access to the extended ASCII character set.

When you enable the number pad, you also turn the **Num Lock** key off or on. When **Num Lock** is on, the number pad keys produce only numbers or extended ASCII characters with the same scan codes as the numeric keypad on a PC. The numeric keys always generate the same ASCII characters for numbers, but the scan codes depend on the status of the number pad and the **Num Lock** setting.

When **Num Lock** is off, the number pad keys move the cursor (**Home**) or edit text (**Del**). Refer to your JANUS user's manual for more information on the number pad.

Syntax:

```
#include <im20lib.h>
IM_STATUS im_number_pad_on
    (IM_NUMBER_LOCK numlock);
```

IN Parameters: The *numlock* parameter indicates whether the **Num Lock** key is ON or OFF. Use one of these constants:

IM_NUMLOCK_ON	Enable the number pad with Num Lock ON
IM_NUMLOCK_OFF	Enable the number pad with Num Lock OFF

OUT Parameters: None.

Return Value: This function returns one of the standard status codes defined in Appendix A, "Status Codes."

See Also: im_number_pad_off

Example

```
/******************* im_number_pad_on  *****************************/
/*Also with: im_number_pad_off */

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <ctype.h>
#include "im20lib.h"

void main (void)
{
int inchar;

   /* Numlock control */
   printf("\nTurn numlock ON/OFF\n");
   printf(“Enter numbers.\n”);
   printf("'Q' to quit\n");

   /* Number Lock On */
   /* IM_NUMLOCK_ON : Enables numeric pad with NUMLOCK ON */
   printf ("Numpad enable NL ON\n");
   inchar = '\0';

   im_number_pad_on( IM_NUMLOCK_ON);

   while (inchar != 'Q')
   {
      inchar = toupper( getche());
   }

   /* IM_NUMLOCK_OFF : Enables numeric pad with NUMLOCK off */
   printf ("\nNumpad enable NL OFF\n");
   printf(“Try to enter numbers.\n”);
   printf("'Q' or ‘3’ to quit\n");
   inchar = '\0';

   im_number_pad_on( IM_NUMLOCK_OFF);

   while (inchar != 'Q')
   {
      inchar = toupper( getche());
   }
}
```

im_parse_host_response

Purpose: This function parses the host response for a DCM transaction. For a detailed discussion of programming database transactions, see Chapter 5, "Using the PSK With DCM."

Syntax:

```
#include <im20lib.h>
IM_USHORT im_parse_host_response
    (IM_USHORT max_ret_length;
    IM_UCHAR *ret_string;
    IM_DCMSTRN ret_array[];
    IM_UCHAR oper_type);
```

IN Parameters: The *max_ret_length* parameter sets the maximum length for the return string buffer.

The **ret_string* parameter is a pointer to the return string buffer. The buffer contains the database server's transaction message unless a timeout occurs.

If the reader receives IM_DCM_TIMEOUT, then the communication error status is placed in the first two bytes of the return buffer.

The return string buffer must be large enough to hold a complete packet from the controller.

OUT Parameters: The *ret_array[]* parameter is the array containing pointers to the variables that hold the data returned from the database. The received field data is parsed and then placed into the variables.

The **oper_type* parameter is a pointer to the operation type and is one of these constants:

IM_DB_READ	R	Read records from the database
IM_DB_INSERT	W	Write records to the database
IM_DB_UPDATE	U	Update records in the database
IM_DB_DELETE	D	Delete records from the database
IM_DB_UNKNOWN	X	Unknown

Return Value: This function returns one of the status codes from the following table. For more information on these return codes, see Chapter 5, "Using the PSK With DCM."

IM_DCM_ACK_RCD	1A01	Transaction successful, ACK received
IM_DCM_MSG_RCD	1A10	Data received
IM_DCM_NAK_RCD	5A02	Transaction failed, NAK received
IM_DCM_TRUNCATED	5A11	Received data was truncated
IM_DCM_EXCESSFIELD	5A12	Too many fields
IM_DCM_TIMEOUT	9A04	Receive timeout
IM_DCM_HEAD_ERR	9A05	Packet header error

Notes: See Chapter 5, "Using the PSK With DCM."

See Also: im_setup_trx, im_standard_trx

Example

```
See the sample program in Chapter 5, "Using the PSK With DCM."
```

im_power_status

Purpose: This function returns the line status, battery status, backup status, and percentage of remaining battery life.

Syntax:

```
#include <im20lib.h>
IM_STATUS im_power_status
    (IM_LINE_STATUS *line_status,
    IM_BATTERY_STATUS battery_status,
    IM_BACKUP_STATUS *backup_status,
    IM_UCHAR fuel_gauge);
```

IN Parameters: None.

OUT Parameters: The *line_status* parameter returns one of these constants:

IM_ACline_NOT_CONNECTED	AC line is not connected
IM_ACline_CONNECTED	AC line is connected
IM_UNKNOWN_ACline	AC line status is unknown

The *battery_status* parameter returns one of these constants:

IM_HIGH_BAT	Battery charge is high
IM_LOW_BAT	Battery charge is low
IM_CRITICAL_BAT	Battery charge is critical
IM_CHARGING_BAT	Battery is charging now
IM_UNKNOWN_BAT	Battery condition is unknown

The *backup_status* parameter returns one of these constants:

IM_BACKUP_OK	Backup battery is OK
IM_BACKUP_LOW	Backup battery is low

The *fuel_gauge* parameter returns one of these constants:

0 to 100	% of full charge in increments of 10%
255	Unknown

Return Value: This function returns one of the standard status codes defined in Appendix A, “Status Codes.”

Example

```
/******************* im_power_status ****************************/
#include <stdio.h>
#include <stdlib.h>
#include "im20lib.h"

IM_LINE_STATUS    ac_line;
IM_BATTERY_STATUS battery;
IM_BACKUP_STATUS  backup;
IM_UCHAR          fuel;
IM_STATUS         status;

void main (void)
{
   /* Call Intermec function */
   status = im_power_status( &ac_line, &battery, &backup, &fuel);
   printf ("Status: %4X\n", status);

   printf ("Line status:\n  ");
   switch (ac_line)
   {
     case IM_ACline_NOT_CONNECTED:
          printf ("Not connected\n");
          break;

     case IM_ACline_CONNECTED:
          printf ("Connected\n");
          break;

     case IM_UNKNOWN_ACline:
          printf ("Unknown\n");
          break;

     default:
          printf ("Invalid return value\n");
          break;
   } /* End switch */

   printf ("Battery status:\n  ");
   switch (battery)
   {
     case IM_HIGH_BAT:
           printf ("Charge high\n");
           break;

     case IM_LOW_BAT:
          printf ("Charge low\n");
          break;

     case IM_CRITICAL_BAT:
          printf ("Charge critical\n");
          break;

     case IM_CHARGING_BAT:
          printf ("Charging battery\n");
          break;

     case IM_UNKNOWN_BAT:
          printf ("Unknown\n");
          break;
```

```
     default:
          printf ("Invalid return value\n");
          break;
   } /* end switch */

printf ("Backup status:\n  ");
   switch (backup)
   {
     case IM_BACKUP_OK:
          printf ("Backup battery OK\n");
          break;

     case IM_BACKUP_LOW:
          printf ("Backup battery low\n");
          break;

     default:
          printf ("Invalid return value\n");
          break;
   } /* end switch */

   if (fuel <= 100)
      printf ("Fuel gauge:\n %u percent\n", fuel);
   else if (fuel == 255)
      printf ("Fuel gauge unknown\n");
   else
      printf ("Invalid return value\n");
}
```

im_protocol_extended_status

Purpose: This function gets the protocol extended status from the designated communications port. You allocate the protocol status buffer or RF status buffer, and then im_protocol_extended_status returns the status and setup in hexadecimal values.

Syntax:

```
#include <im20lib.h>
IM_STATUS im_protocol_extended_status
    (IM_COM_PORT port_id,
    IM_COMM_STATUS_BUFFER_s far
    *status_buffer);
```

IN Parameters: The *port_id* parameter identifies the communications port as follows:

IM_COM1	COM1
IM_COM2	COM2, scanner port (2010 and 2050)
IM_COM4	COM4 (RF only)

The *status_buffer* parameter is a far pointer to the status buffer structure IM_COMM_STATUS_BUFFER_s.

OUT Parameters: None.

Return Value: This function returns one of the standard status codes defined in Appendix A, "Status Codes."

See Also: im_receive_buffer_no_wait, im_transmit_buffer_no_wait

Example

```
/******************* im_protocol_extended_status *********************/
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#include <im20lib.h>

void main (void)
{
IM_STATUS               status;
IM_COMM_STATUS_BUFFER_s status_buffer;

   /* First format string for extended protocol status */
   IM_UCHAR *format_str1 =
      "Strut Ver = %d\nHd Ver = %s\nBaud = %lu\nParity=%XH\nStop=%XH\nData=%XH\n"
      "Modem=%XH\nPort stat = %XH\nActive prot = %d\n"
```

```
      "Lrc enabled = %XH\nInterchar dely= %d\nTurnarnd dely=%d\n"
      "Receive tmeout= %d\nTransmit tmeout= %d\n";

/* Second format string for extended protocol status */
   IM_UCHAR *format_str2 =
      "Pol char = %XH\nPol length= %d\nSel char = %XH\nSel length= %d\n"
      "Res char = %XH\nRes length= %d\nReq char = %XH\nReq length= %d\n"
      "Aff char = %XH\nAff length= %d\nNeg char = %XH\nNeg length= %d\n"
      "Som char = %XH\nSom length= %d\n";

   /* Third format string for extended protocol status */
   IM_UCHAR *format_str3 =
      "Tx eom len = %XH\nTx eom char 1 = %XH\n"
      "Tx eom char 2 = %XH\nRx eom len = %XH\nRx eom char 1 = %XH\n"
      "Rx eom char 2 = %XH\nFlow ctrl = %XH\nMulti drp addr = %XH\n"
      "Multi drp addr = %XH\nMulti drp enbled = %XH\n"
      "EOF len = %XH\nEOF char = %XH\nRecv clt buff= %d\n"
      "Xmit clt buff= %d\nProt mode = %d\n";

   /* Call Intermec routine */
   status_buffer.status_structure_version = STATUS_STRUCT_VERSION;

   status = im_protocol_extended_status(IM_COM1, &status_buffer);

   clrscr();
   gotoxy(1, 1);     /* to upper left screen */

   /* Print first page of status info */
   printf("Extd Status,Page 1of3\n");
   printf(format_str1,
           status_buffer.status_structure_version,
           status_buffer.handler_ver,

           status_buffer.specific.serial.baud_rate,
           status_buffer.specific.serial.parity,
           status_buffer.specific.serial.stop_bits,
           status_buffer.specific.serial.data_bits,
           status_buffer.specific.serial.modem_status,
           status_buffer.specific.serial.port_status,
           status_buffer.specific.serial.active_prot,
           status_buffer.specific.serial.lrc_enabled,
           status_buffer.specific.serial.interchar_delay,
           status_buffer.specific.serial.turnaround_delay,
           status_buffer.specific.serial.receive_timeout,
           status_buffer.specific.serial.transmit_timeout);

   /* Pause between screens */
   printf("Any key to cont'\n");
   getch();

   clrscr();
   gotoxy(1, 1);
   printf ("Extd Status,Page 2of3\n");

   /* Print second page of status info */
   printf (format_str2,

            status_buffer.specific.serial.pol_char,
            status_buffer.specific.serial.pol_length,
            status_buffer.specific.serial.sel_char,
            status_buffer.specific.serial.sel_length,
            status_buffer.specific.serial.res_char,
```

```
            status_buffer.specific.serial.res_length,
            status_buffer.specific.serial.req_char,
            status_buffer.specific.serial.req_length,
            status_buffer.specific.serial.aff_char,
            status_buffer.specific.serial.aff_length,
            status_buffer.specific.serial.neg_char,
            status_buffer.specific.serial.neg_length,
            status_buffer.specific.serial.som_char,
            status_buffer.specific.serial.som_length);

   /* Pause between screens */
   printf ("Any key to cont'\n");
   getch();

   clrscr();
   gotoxy(1, 1);
   printf ("Extd Status,Page 3of3\n");

   /* Print third page of status info */
   printf (format_str3,
            status_buffer.specific.serial.txeom_len,
            status_buffer.specific.serial.txeom_char_1,
            status_buffer.specific.serial.txeom_char_2,
            status_buffer.specific.serial.rxeom_len,
            status_buffer.specific.serial.rxeom_char_1,
            status_buffer.specific.serial.rxeom_char_2,
            status_buffer.specific.serial.flow_ctrl,
            status_buffer.specific.serial.multi_drop_addr,
            status_buffer.specific.serial.multi_drop_enabled,
            status_buffer.specific.serial.eof_length,
            status_buffer.specific.serial.eof_char,
            status_buffer.specific.serial.have_recv_client_buffer,
            status_buffer.specific.serial.have_xmit_client_buffer,
            status_buffer.specific.serial.protocol_mode);

   /* Function call return status */
   printf ("Extd Prot Stat=%XH\n", status);
}
```

im_ready_for_reboot

Purpose: This function indicates to the host whether the JANUS reader is ready for a reboot in response to having received a "Prepare for Reboot" command.

Syntax:
```
#include "im20lib.h"
IM_STATUS im_ready_for_reboot(IM_BOOL reboot_flag)
```

IN Parameters: The *reboot_flag* parameter can be one of the following symbolic constants:

IM_TRUE	JANUS reader is ready for reboot. The string "..+%1" will be sent to the host.
IM_FALSE	JANUS reader is not ready for reboot. The string "..+%0" will be sent to the host.

OUT Parameters: None.

Return Value: This function returns one of the standard status codes defined in Appendix A, "Status Codes."

Notes: To reboot, the JANUS reader must receive one of the reboot commands from the host: ..%$0 to reboot without an open session, or ..%$1 to reboot with an open session.

The im_ready_for_reboot function with a *reboot_flag* of IM_TRUE **must** be called before the JANUS reader will respond to either reboot command. This prevents the JANUS reader from being rebooted before it is ready.

See Also: im_get_reboot_flag

Example

```
See example for im_get_reboot_flag
```

im_receive_buffer

Purpose: This function receives the contents of a data buffer from a designated communications port.

Syntax:

```
#include <im20lib.h>
IM_STATUS im_receive_buffer
    (IM_COM_PORT port_id,
    IM_USHORT user_length,
    IM_UCHAR far *data_buffer,
    IM_USHORT timeout,
    IM_USHORT far *comm_length);
```

IN Parameters: The *port_id* parameter identifies the communications port as follows:

IM_COM1	COM1
IM_COM2	COM2, scanner port (2010 and 2050)
IM_COM4	COM4 (RF only)

The *user_length* parameter is the maximum number of bytes to receive and must be at least 256 bytes.

The *data_buffer* parameter is a far pointer to the data array where you want to place the received data. This buffer must hold at least 256 bytes.

The *timeout* parameter is the receive timeout period. You can exit the function before data is received or before the timeout occurs by performing an application break sequence (see your JANUS user's manual). The return status indicates whether the function was successful, a timeout occurred, or the application break was received.

The *timeout* parameter is a number or one of these constants:

1 to 65,534 ms	Numeric range
IM_ZERO_TIMEOUT	No wait
IM_INFINITE_TIMEOUT	Wait forever

If you select IM_INFINITE_TIMEOUT, the function will not return until the end of message character has been received.

OUT Parameters: The *comm_length* is the actual number of bytes received upon completion of the call.

Return Value: This function returns one of the standard status codes defined in Appendix A, "Status Codes."

Notes: This function does not return until an end of message, a buffer is full, a timeout, or an error occurs.

You must install a protocol handler to use this function.

See Also: im_receive_buffer_no_wait, im_receive_buffer_noprot, im_cancel_rx_buffer, im_rx_check_status

---

Example

```
/******************* im_receive_buffer ******************************/
/* Also with: im_cancel_rx_buffer */
/*                                */
/* Note: Must have reader services and comm protocol handler installed to  */
/*       enable send/receive buffer functions.                             */
/*       Run these TSR's at DOS prompt before running these tests:         */
/*        rservice                                                         */
/*        phimec 1  ( 1 is COM1)                                           */
/*                                                                         */
/*       Assume that the receive data is stored in file 'RC.DAT' at current*/
/*       working directory and uses the first byte of receiving string to  */
/*       terminate receive buffer loop.  Default is 'Q' or 'q' character   */

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "im20lib.h"

static IM_COM_PORT  com_port = IM_COM1;      /* Uses COM1*/
static IM_UCHAR     filename[] = "RC.DAT";   /* Store receive data filename*/
static IM_UCHAR     eor_char   = 'Q';        /* Default end of receive char */

void main (void)
{
IM_USHORT     actual_length;
IM_UCHAR      data_buffer[300];
IM_UCHAR      eor_byte;
FILE          *out_fptr;
IM_STATUS     status;

   /* Call im_cancel_Rx_buffer to initialize the communications port and its buffer */
   status = im_cancel_rx_buffer(com_port);
   printf("Cancel Rx status: %xH\n", status);

   /* Assume that the received data is stored in file 'RC.DAT' */
   out_fptr = fopen(filename, "w+t");         /* Open in Translate mode*/

   printf("Type End of receive\nchar?");
   eor_byte = toupper( getche());             /* Get End of Receive char */
   if (isascii(eor_byte))
```

```
      eor_char = eor_byte;

   printf ("Enter characters\n");
   printf ("to be received\n");
   printf ("from host\n");
   printf ("CRLF to End of Line\n");    /* Each line end with a CR & LF */

   do
   {
      actual_length = 0;

      /* Call Intermec receive buffer function with 50000 msec timeout */
      status = im_receive_buffer(com_port, sizeof(data_buffer) - 1,
                                 data_buffer, 50000, &actual_length);

      if (actual_length > 0)
      {
         /* Append CR & LF before write it into files */
         data_buffer[actual_length++] = '\x0D';
         data_buffer[actual_length++] = '\x0A';
         data_buffer[actual_length] = '\0';

         /* Write results to file */
         fwrite(data_buffer, actual_length, sizeof(char), out_fptr);

         printf("%s", data_buffer);        /* Write data to on screen */
         eor_byte = toupper(data_buffer[0]);
      }
      else
      {
         eor_byte = '\0';
         printf("Rx Status: %xH\n", status);
      }

   }  while (eor_byte != eor_char);       /* Loop until EOR char 'q' */

   /* Close file and exit */
   fclose(out_fptr);
   printf("Receive Done\n");
}
```

im_receive_buffer_no_wait

Purpose: This function receives data from the designated communications port and places the data into a user-defined record. It differs from im_receive_buffer in that the programmer must set up the data record and monitor the transfer status until transmission is complete. This function runs in the background after it is initiated; no checks are made.

Use this function when receiving multiple buffer transmissions in conjunction with im_rx_check_status.

Syntax:

```
#include <im20lib.h>
IM_STATUS im_receive_buffer_no_wait
    (IM_COM_PORT port_id,
    IM_COM_DATA_BUFFER far *data_struct);
```

IN Parameters: The *port_id* parameter identifies the communications port as follows:

IM_COM1	COM1
IM_COM2	COM2, scanner port (2010 and 2050)
IM_COM4	COM4 (RF only)

The *data_struct* parameter is a far pointer to the data array for the received data. This buffer must hold at least 256 bytes.

OUT Parameters: None.

Return Value: This function returns one of the standard status codes defined in Appendix A, “Status Codes.”

Notes: You must build and maintain a *data_struct* of the type IM_COM_DATA_BUFFER as described in IM20LIB.H.

You must periodically check the protocol_xfer_status element in the IM_COM_DATA_BUFFER record to see if all the data has been received. When protocol_xfer_status is no longer code 8602H (communications port in use), all data is received.

You must install a protocol handler to use this function.

This function differs from im_receive_buffer_noprot in that program execution continues after im_receive_buffer_no_wait is called. When im_receive_buffer_noprot returns from a function call, the data has been read, the timeout has expired, or an error was encountered.

See Also: im_receive_buffer_noprot, im_cancel_rx_buffer, im_rx_check_status

---

Example

```
/******************* im_receive_buffer_no_wait ***********************/
/* Also with: im_rx_check_status */
/*   im_cancel_rx_buffer         */
/*                               */
/* Note: Must have reader services and comm protocol handler installed to  */
/*       enable send/receive buffer functions.                             */
/*       Run these TSR's at DOS prompt before running these tests:         */
/*        rservice                                                         */
/*        phimec 1  ( 1 is COM1)                                           */

#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
#include "im20lib.h"
#include "immsg.h"

static IM_COM_PORT  com_port = IM_COM1;    /* Uses COM1*/
static IM_UCHAR     eor_char   = 'Q';      /* Default end of receive char */

void main (void)
{
IM_COM_DATA_BUFFER  com_buff_struct; /*Data buffer structure for receive/transmit */
IM_STATUS           status;
IM_UCHAR            eor_byte;
IM_UCHAR            data_buff[300];
int                 index = 0;

printf("Type End of receive\nchar?");
   eor_byte = getche();                /* Get End of Receive char */
   eor_byte = toupper( eor_byte);

   if ( isascii(eor_byte))
      eor_char = eor_byte;

   printf ("Enter characters\n");
   printf ("to be received\n");
   printf ("from host\n");
   printf ("CRLF to End of Line\n");    /* Each line end with a CR & LF */

   /* Get the receive environment ready */
   status = im_cancel_rx_buffer(com_port);
   printf ("RX Init =%XH\n", status);

   /* Set up com_buff structure for init Receive Buffer No Wait */
   com_buff_struct.command       = IM_CU_RECEIVE;
   com_buff_struct.comm_length   = 0;
```

```
   com_buff_struct.user_length   = sizeof(data_buff);
   com_buff_struct.data_ptr      = data_buff;
   com_buff_struct.protocol_mode = IM_NO_PROTOCOL;
   com_buff_struct.eom_char      = 0x0d;        /* CR for terminating char */
   com_buff_struct.protocol_xfer_status = IM_COMM_INUSE;

   /* Call Intermec function */
   status = im_receive_buffer_no_wait(com_port, (IM_COM_DATA_BUFFER far *)
                                              &com_buff_struct);

   /* Uses im_rx_check_status to get the rest coming data */
   do        /* loop until first character in buffer is a 'Q' */
   {
      /* Check for receive errors: Print the error status */
      if ( IM_ISSUCCESS(status) )
      {
         ++index;            /* Indicator of the receiving buffer */

         /* Let the receive data coming complete */
         while (com_buff_struct.protocol_xfer_status == IM_COMM_INUSE)
            ;    /* End of While loop, Wait the coming data settled */

         printf ("Line %d Done\n", index);

         /* Put a null byte in received string for display */
         data_buff[com_buff_struct.comm_length] = '\0';

         /* If 1st byte in string is equal to eor_char, then quit the receive*/
         eor_byte = toupper(data_buff[0]);

         /* Display received string to screen */
         printf("%s\n", data_buff);

         /*
          ************************************************************
          * Update buffer for next block before calling check status to
          * get protocol handler to reset status.
          ************************************************************
          */
         com_buff_struct.command              = IM_CU_RECV_AGAIN;
         com_buff_struct.comm_length          = 0;
         com_buff_struct.protocol_xfer_status = IM_COMM_INUSE;

         /* Call intermec function to see next block data */
         status = im_rx_check_status(com_port);
      }
      else
      {
         eor_byte = '\0';
         printf("Rx Error = %XH\n", status);
      }

printf("ch: %xH, by: %xH\n", eor_char, eor_byte);

   }  while ( eor_byte != eor_char);   /* Loop until reaching eor_char */

   printf ("Receive Done\n");
}
```

im_receive_buffer_noprot

Purpose: This function receives a text string from the designated port without using the active protocol. Only the PHIMEC.EXE protocol handler supports this function, which is often used to receive a host initialization string before beginning transmission.

Syntax:

```
#include <im20lib.h>
IM_STATUS im_receive_buffer_noprot
    (IM_COM_PORT port_id,
    IM_USHORT user_length,
    IM_UCHAR far *data_buffer,
    IM_USHORT timeout,
    IM_UCHAR eom_char,
    IM_USHORT far *comm_length);
```

IN Parameters: The *port_id* parameter identifies the communications port as follows:

IM_COM1	COM1
IM_COM2	COM2, scanner port (2010 and 2050)
IM_COM4	COM4 (RF only)

The *user_length* parameter is the maximum number of characters you can receive and must be at least 256 bytes.

The *data_buffer* parameter is a far pointer to the data array where you want to place the received data. This buffer must hold at least 256 bytes.

The *timeout* parameter is the receive timeout period. You can exit the function before data is received or before the timeout occurs by performing an application break sequence (see your JANUS user's manual). The return status indicates whether the function was successful, a timeout occurred, or the application break was received.

The *timeout* parameter is a number or one of these constants:

1 to 65,534 ms	Numeric range
IM_ZERO_TIMEOUT	No wait
IM_INFINITE_TIMEOUT	Wait forever

If IM_INFINITE_TIMEOUT is selected, the function will not return until the end of message character has been received.

The *eom_char* parameter is the end of message character needed to terminate input from the communications port. The character is normally a carriage return, but you can set it to any character.

OUT Parameters: The *comm_length* parameter returns the actual number of bytes received.

Return Value: This function returns one of the standard status codes defined in Appendix A, "Status Codes."

Notes: You must install the PHIMEC.EXE protocol handler to use this function.

The difference between this function and im_receive_buffer_no_wait is that when im_receive_buffer_noprot returns from the function call, the data has been read, the timeout has expired, or an error was encountered. When you call im_receive_buffer_no_wait, program execution continues without checking for received data or errors.

See Also: im_receive_buffer_no_wait, im_cancel_rx_buffer, im_rx_check_status

Example

```
/******************* im_receive_buffer_no_protocol ***********************/
/* Also with: im_cancel_rx_buffer  */
/*                                 */
/* Note: Must have reader services and comm protocol handler installed to  */
/*       enable send/receive buffer functions.                             */
/*       Run these TSR's at DOS prompt before running these tests:         */
/*        rservice                                                         */
/*        phimec 1  ( 1 is COM1)                                           */

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include "im20lib.h"
#include "immsg.h"

static IM_COM_PORT  com_port = IM_COM1;      /* Uses COM1*/
static IM_UCHAR     eor_char   = 'Q';       /* Default end of receive char */

void main (void)
{
IM_STATUS status;
IM_UCHAR  eor_byte;        /* Uses 1st byte of receive to determine quit Rx*/
IM_UCHAR  eom_char;        /* End of message char */
IM_USHORT actual_length;
IM_UCHAR  data_buffer[300];

/* Get the receive environment ready */
   status = im_cancel_rx_buffer(com_port);
```

```
   printf("Rx Cancel: %xH\n", status);
   printf("Type End of receive\nchar?");
   eor_byte = toupper( getche());           /* Get End of Receive char */
   if (isascii(eor_byte))
      eor_char = eor_byte;

   printf ("Enter characters\n");
   printf ("to be received\n");
   printf ("from host\n");
   printf ("CRLF to End of Line\n");    /* Each line end with a CR & LF */

/* Uses Carriage Return for End of Message character */
   eom_char = 0x0d;         /*CR */

   do
   {
      /* Call the Intermec function */
      status = im_receive_buffer_noprot(com_port, sizeof(data_buffer),
                                        (char far *)data_buffer, 50000,
                                        eom_char, &actual_length);

      if (actual_length > 0)
      {
         /* append CR & LF before write it into files */
         data_buffer[actual_length++] = '\x0D';
         data_buffer[actual_length++] = '\x0A';
         data_buffer[actual_length] = '\0';

         printf("%s", data_buffer);        /* Write data to on screen */
         eor_byte = toupper(data_buffer[0]);
      }
      else
      {
         eor_byte = '\0';
         printf("Rx Status: %xH\n", status);
      }

   }  while (eor_byte != eor_char);  /* Loop until reaching End of Receive*/

   /* Show Exit receive no protocol message */
   printf("Rx No Protocol Done\n");
}
```

im_receive_byte

Purpose: This function receives one byte of data through the designated communications port. This function is identical to MS-DOS INT 14H service 02H.

Syntax:
```
#include <im20lib.h>
IM_STATUS im_receive_byte
    (IM_COM_PORT port_id,
    IM_UCHAR *receive_byte);
```

IN Parameters: The *port_id* parameter identifies the communications port as follows:

IM_COM1	COM1
IM_COM2	COM2, scanner port (2010 and 2050)
IM_COM4	COM4 (RF only)

OUT Parameters: The *receive_byte* parameter is a pointer to the byte received from the communications port.

Return Value: This function returns one of the standard status codes defined in Appendix A, “Status Codes.”

Notes: This function requires the PC standard protocol handler PHPCSTD.EXE.

If you are using PHIMEC.EXE and want to receive a single byte, use im_receive_buffer with a buffer length of one instead of the im_receive_byte procedure.

This procedure will not work if linked.

See Also: im_transmit_byte

Example

```
/****************** im_receive_byte ****************************/
/* Uses Borland _bio_serialcom or Microsoft _bios_serialcom  to init the   */
/* com port for single byte receive.                                       */
/* Operations. It sets up for 9600,E,7,1 on serial port                    */
/* To Terminate reception of bytes:                                        */
/*    To end the reception of bytes from the host, type <ESC> on JANUS     */
/*     and then type a Ctrl Z on the host                                  */
/* Note: Phimec protocol handler must NOT be installed to run this routine */
/*       Phpcstd protocol handler must be installed to run this routine    */
```

```
#include <stdio.h>
#include <dos.h>
#include <conio.h>
#include <stdlib.h>
#include <bios.h>
#include <string.h>
#include "im20lib.h"
#include "immsg.h"

/* Byte oriented defines */
#define COM1       0
#define SETTINGS (_COM_9600 | _COM_CHR7 | _COM_STOP1 | _COM_EVENPARITY )

static IM_COM_PORT  com_port = IM_COM1;      /* Uses COM1*/

void main (void)
{
IM_UCHAR  inchar;
IM_STATUS status;

printf("\nChoose a protocol:\n");
   printf("1) PC Standard\n");
   printf("2) No protocol\n");

   if (getche() == '2')       /* Call bios to initialize com port */
   {
      /* Communications port init */
      _bios_serialcom(_COM_INIT, COM1, SETTINGS);
   }

   printf ("\nChars received\n");
   printf ("<ESC> to quit\n");
   printf ("PC function\n");
   printf ("Ctrl Z to finish at\n");
   printf ("host\n");

/* Phimec protocol handler must NOT be installed ...*/
   while (1)
   {
      /* Call Intermec function */
      status = im_receive_byte(com_port, &inchar);

      if ( IM_ISERROR(status))
      {
         printf("st: %xH\n", status);
         im_message(status);
      }
      else
         putch (inchar);

      /* If User press <ESC> key, then stop program */
      if ( kbhit() && getch() == '\x1B')
         break;
   }
}
```

im_receive_input

Purpose: This function gets input from the source and places it into the received buffer. You can use the im_get_length function after this function to get the input length.

Syntax:

```
#include <im20lib.h>
IM_STATUS im_receive_input
    (IM_ORIGIN allowed,
    IM_UINT timeout,
    IM_ORIGIN *source,
    IM_CHAR *received);
```

IN Parameters: The *allowed* parameter defines the available input source and is one or more of these constants:

IM_LABEL_SELECT	Label selected
IM_KEYBOARD_SELECT	Keypad selected
IM_COM1_SELECT	COM1 selected
IM_COM2_SELECT	COM2, scanner port selected (2010 and 2050)
IM_COM4_SELECT	COM4 selected (RF only)
IM_ALL_SELECT	All sources are selected

The *timeout* parameter is the receive timeout period. You can exit the function before data is received or before the timeout occurs by performing an application break sequence (see your JANUS user's manual). The return status indicates whether the function was successful, a timeout occurred, or the application break was received.

The *timeout* parameter is a number or one of these constants:

1 to 65,534 ms	Numeric range
IM_ZERO_TIMEOUT	No wait
IM_INFINITE_TIMEOUT	Wait forever

If IM_INFINITE_TIMEOUT is selected, the function will not return until the end of message character has been received.

OUT Parameters: The *source* parameter specifies the actual input source and is one of these constants:

IM_NO_SELECT	No selection made
IM_LABEL_SELECT	Label selected
IM_KEYBOARD_SELECT	Keypad selected
IM_COM1_SELECT	COM1 selected
IM_COM2_SELECT	COM2, scanner port selected (2010 and 2050)
IM_COM4_SELECT	COM4 selected (RF only)

The *received* parameter is the variable where the data is placed.

Return Value: This function returns one of the standard status codes defined in Appendix A, "Status Codes."

Notes: If input from more than one source is received before this function is called, the first available input is returned in this order: label, keypad, COM1, COM2, and COM4.

If you are using a communications port, you must install a protocol handler and call im_link_comm.

You must install the Reader Wedge to use this function. For information on loading RWTSR.EXE, see "Runtime Requirements" earlier in this chapter.

See Also: im_get_length, im_get_label_symbology, im_link_comm

---

Example

```
/******************* im_receive_input ******************************/
/* Also with: im_get_length */
/*   im_link_comm           */
/*   im_unlink_comm         */
/*   im_get_input_mode      */
/*   im_set_input_mode      */

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include "im20lib.h"
#include "immsg.h"

#define COM_BUFSIZE 300            /* Allocate 300 bytes comm buffer */

void main (void)
{
IM_UCHAR   input[300];
```

```
IM_USHORT  length;
IM_MODE    original_mode;
IM_UCHAR   *com_buffer;
IM_ORIGIN  in_source;
IM_ORIGIN  source;
IM_STATUS  status;

   window(1, 1, 20, 16);       /* Set the application text window */
   clrscr();                   /* Clear the screen  */

   printf("Demo im_receive_input\n'Q' to quit\n\'C' to clear screen\n");

   /* Allocate a comm buffer */
   com_buffer = (char *) malloc(COM_BUFSIZE);

   /* Get the previous input mode for restoring input mode later */
   original_mode = im_get_input_mode();

   /* Set Reader Wedge mode to program interface */
   im_set_input_mode(IM_PROGRAMMER);

   /* Link com_buffer to com1 */
   im_link_comm(IM_COM1, com_buffer, COM_BUFSIZE);

   /* Input loop  */
   do
   {
      /* Set up input source */
      source = IM_LABEL_SELECT | IM_KEYBOARD_SELECT | IM_COM1_SELECT;

      /* Request input from label, keypad or com1 */
      status = im_receive_input(source, IM_INFINITE_TIMEOUT, &source, input);
      length = im_get_length(source);

      if (IM_ISGOOD(status))
      {
         /* Show the input source */
         if (source == IM_LABEL_SELECT)
            printf("\nLabel input:\n");
         else if (source == IM_KEYBOARD_SELECT)
            printf("\nKeybd input:\n");
         else if ( source == IM_COM1_SELECT)
            printf("\nCom1 input:\n");
         /* Display input data */
         printf("%s\nInput length: %d\n", input, length);
      }
      else if (status == IM_APPLICATION_BREAK)
      {
         printf("application break detected");
      }
      else /* input error */
         printf("input error\n");

      /* Upper case first char of input for simplifying to test input */
      input[0] = toupper(input[0]);

      /*If the first char in string is 'C', then clear screen.*/
      if (input[0] == 'C')
         clrscr();

   } while (input[0] != 'Q');  /* First number in string is 'Q', then stop */
```

```
   /* Unlink com_buffer from com1 */
   im_unlink_comm(IM_COM1);

   /* Free allocated memory */
   free(com_buffer);

   /* Restore original input mode */
   im_set_input_mode(original_mode);
}
```

im_rs_installed

Purpose: This function determines whether or not the reader services (RSERVICE.EXE) program is installed.

Syntax:
```
#include <im20lib.h>
IM_STATUS im_rs_installed (void);
```

IN Parameters: None.

OUT Parameters: None.

Return Value: This function returns one of the standard status codes defined in Appendix A, “Status Codes.”

See Also: None.

Example

```
/******************* im_rs_installed ******************************/
/* Also with: im_message */

#include <stdio.h>
#include "im20lib.h"

void main (void)
{
   printf("Check for reader\nservice installation\n");
   im_message( im_rs_installed() );
}
```

im_rx_check_status

Purpose: This function directs the active protocol handler to check the communications port buffer status variable to determine if the application program has accepted the previous data. Use this function with the im_receive_buffer_no_wait function to receive input from multiple buffers.

Syntax:

```
#include <im20lib.h>
IM_STATUS im_rx_check_status
    (IM_COM_PORT port_id);
```

IN Parameters: The *port_id* parameter identifies the communications port as follows:

IM_COM1	COM1
IM_COM2	COM2, scanner port (2010 and 2050)
IM_COM4	COM4 (RF only)

OUT Parameters: None.

Return Value: This function returns one of the standard status codes defined in Appendix A, "Status Codes."

Notes: You must install a protocol handler to use this function.

See Also: im_receive_buffer_no_wait

Example

```
See example for im_receive_buffer_no_wait.
```

im_serial_protocol_control

Purpose: This function controls the protocol mode for a designated communications port and determines whether the receive mode is with or without protocol according to the input parameter. Any change clears the input buffer and resets the reader command parser.

Use this function to set the protocol when connected to a communications port. The return value indicates a success if the port is currently linked to the Virtual Wedge. If the port is not currently linked to the Virtual Wedge, the function returns an error.

Syntax:

```
#include <im20lib.h>
IM_STATUS im_serial_protocol_control
    (IM_COM_PORT port_id,
    IM_PROTOCOL_CMD protocol,
    IM_UCHAR eom_char);
```

IN Parameters: The *port_id* parameter identifies the communications port as follows:

IM_COM1	COM1
IM_COM2	COM2, scanner port (2010 and 2050)
IM_COM4	COM4 (RF only)

The *protocol* parameter affects how incoming messages are received and is one of these constants:

IM_NO_CHANGE	Keep the current protocol
IM_PROTOCOL_OFF	Turn the protocol OFF
IM_PROTOCOL_ON	Turn the protocol ON
IM_NEW_TERM_CHAR	Use the new termination character

The *protocol* parameter affects all incoming messages as follows:

- If IM_NO_CHANGE is specified, the protocol and end of message settings remain the same as the last time the function was called.
- If IM_PROTOCOL_OFF is specified, the incoming data is terminated with the specified end of message character. If the end of message character is null, the function terminates upon receiving the first character.

- If IM_PROTOCOL_ON is specified, the incoming message is terminated with the end of message character of the active protocol.
- If IM_NEW_TERM_CHAR is specified, IM_PROTOCOL_OFF is assumed and the new end of message character specified in the parameter list is used.

The *eom_char* parameter is the end of message character needed to terminate input from the communications port. The character is normally a carriage return, but you can set it to any other character. An EOM character is required when using IM_PROTOCOL_OFF and IM_NEW_TERM_CHAR for *protocol.*

OUT Parameters: None.

Return Value: This function returns one of the standard status codes defined in Appendix A, "Status Codes."

Notes: Requires im_link_comm.

The receive buffers are cleared whenever the protocol is turned ON or OFF. To change the termination character (or to not change the setting at all), use IM_NEW_TERM_CHAR or IM_NO_CHANGE instead of toggling the protocol.

See Also: im_link_comm

Example

```
/******************* im_serial_protocol_control ***********************/
/* Also with: im_set_input_mode  */
/*   im_link_comm                */
/*   im_unlink_comm              */

#include <stdio.h>
#include <conio.h>
#include <dos.h>
#include <stdlib.h>
#include <ctype.h>
#include "immsg.h"
#include "im20lib.h"

#define  COM_BUFSIZE 300        /* Allocate 300 bytes for comm buffer */

void main (void)
{
IM_UCHAR    *com_buffer;
IM_UCHAR    input[256];
```

```
IM_ORIGIN  source;
IM_STATUS   status;
IM_CHAR     *msg_ptr;
int         sw_flag,
            order_flag;

   window(1, 1, 20, 16);      /* Set the reader services text window */
   clrscr();                  /* Clear the screen  */
   printf("Demo IM_SERIAL_PROTOCOL_CONTROL\n'Q' to quit\n");

   /* Allocate a comm buffer */
   com_buffer = (char *) malloc(COM_BUFSIZE);

   /* Set Reader Wedge into PROGRAMMER mode  */
   im_set_input_mode(IM_PROGRAMMER);

   /* Link com_buffer to com1 */
   im_link_comm(IM_COM1, com_buffer, COM_BUFSIZE);

order_flag = 0;           /* Set flag  to start from CR & LF */

   /* Demo Input loop */
   do
   {
      sw_flag = order_flag % 3;
      switch (sw_flag)
      {
        case 0:   /*Turn on protocol handle with EOM char CR & LF. */
           status = im_serial_protocol_control(IM_COM1, IM_PROTOCOL_ON, 0x0D);
           msg_ptr = "CR & LF";
           break;

        case 1:   /* Turn off protocol handle with EOM char Ctrl-Z */
           status = im_serial_protocol_control(IM_COM1, IM_PROTOCOL_OFF, 0x1A);
           msg_ptr = "Ctrl-Z";
           break;

        case 2:   /* Turn off protocol handle with EOM char Ctrl-Y */
           status = im_serial_protocol_control(IM_COM1, IM_PROTOCOL_OFF, 0x19);
           msg_ptr = "Ctrl-Y";
           break;
      }
      printf("\nEOM: %s, P-status: %x\n", msg_ptr, status);
      im_receive_input(IM_COM1, IM_INFINITE_TIMEOUT, &source, input);

      if ( source == IM_COM1_SELECT)
      {
         printf("%s\n", input);
      }
      else
         printf("error\n");
      order_flag++;
      input[0] = toupper(input[0]);

    } while (input[0] != 'Q');

    /* Unlink com_buffer from com1 */
    im_unlink_comm(IM_COM1);
    /* Free allocated memory */
    free(com_buffer);

    im_set_input_mode(IM_WEDGE);
}
```

im_set_abort_callback

Purpose: This function sets up the callback address for notification of an exit program reader command. It determines which function is activated when /$ is received. For example, use this function when you are running a program and want to exit without using a warm boot.

Syntax:

```
#include <im20lib.h>
IM_STATUS im_set_abort_callback
    (IM_ORIGIN source,
    void *notify_routine);
```

IN Parameters: The *source* parameter is one of these constants:

IM_LABEL_SELECT	No source selected
IM_COM1_SELECT	COM1 selected
IM_COM2_SELECT	COM2, scanner port selected (2010 and 2050)
IM_COM4_SELECT	COM4 selected (RF only)
IM_ALL_SELECT	All sources are selected

The *notify_routine* parameter is a pointer to the callback routine.

OUT Parameters: None.

Return Value: This function returns one of the standard status codes defined in Appendix A, “Status Codes.”

Notes: IRL uses the /$ as an exit command when received from a host or scanned from a label. This im_set_abort_callback function allows you to specify a routine that is called whenever the /$ is detected from the host or from the scanner. You can also use this capability from within a PSK application. The /$ is called the IRL exit command, but the application can use it for actions other than exiting the program.

The callback is called by the Reader Wedge as soon as the /$ is detected. The /$ must be the only two characters in the string. Nothing occurs if the /$ is in the middle of other characters. It is recommended that your callback routine be short (such as setting a flag or performing some other action).

Scan in the exit IRL program command to activate the abort callback. When the abort callback routine is called, the processor services an interrupt. The routine must reenable interrupts before doing any other interrupt-driven function such as writing to the screen.

If you use im_set_abort_callback to set up a callback, you must use im_clear_abort_callback before exiting your program. Otherwise, the Reader Wedge continues to call the callback each time the /$ is detected. Since the callback is no longer at the address that the Reader Wedge calls, the result is unpredictable.

The notification *source* parameter replaces the previously set *source*. Only one callback routine can be submitted. On callback, the abort function's first parameter contains the origin of the abort program command. The second parameter is left for further expansion.

This command does not work in Wedge mode.

For this routine to work correctly, you must use the keyword _loadds to set the data segment in the callback routine.

Since the abort routine is called as an interrupt, all other interrupts are disabled until the system returns from the abort function or until an interrupt is explicitly enabled.

You must install the Reader Wedge to use this function. For information on loading RWTSR.EXE, see "Runtime Requirements" earlier in this chapter.

See Also: im_clear_abort_callback

Example

```
/******************* im_set_abort_callback ******************************/
/* Also with: im_clear_abort_callback */
/*   im_set_input_mode                */
/*   im_standby_wait                  */
/*   im_irlv                          */
/*                                    */
/* Note: When using Microsoft "C" compiler, Inside Abort_Callback routines */
/*       avoid to call any run-time routines referring to __iob variable  */
/*       Since microsoft put __iob into another segment. */
/*       Borlandc "C" compiler doesn't have this problem. */

#include <stdio.h>
#include <conio.h>
#include <dos.h>
#include "im20lib.h"
#include "immsg.h"

/*  INTERNAL FUNCTIONS (EXPORTED)   */
void _loadds Abort_Callback(unsigned short arg1, unsigned short arg2);

/*  STATIC FUNCTIONS (KNOWN ONLY BY THIS FILE)  */
static int abort_flag;

void main (void)
{
   IM_UCHAR   input[256];
   IM_ORIGIN  source;
   IM_USHORT  cc;
   IM_DECTYPE symbol;

   /* FUNCTION BODY */
   window(1, 1, 20, 16);       /* Set the application text window */
   clrscr();                   /* Clear the screen  */

   /* Display Demo message */
   printf("Demo IM_SET_ABORT_CALLBACK\n 'Q' to quit\n");

   /* Set Reader Wedge mode to program interface */
   im_set_input_mode(IM_PROGRAMMER);

   /* Set up abort callback notification hook */
   im_set_abort_callback(IM_ALL_SELECT, Abort_Callback);

   /* Clear the abort flag */
   abort_flag  = 0;

/* Demo's input loop */
   do
   {
      /* Test wait */
      printf("Start waiting for 3 seconds\n");
      im_standby_wait(3000);
      printf("Done waiting for 3 seconds\n");

      /* Test input IRL V */
      printf("IRL V test\n");

      /* Set source for labels and keypad */
      source = IM_LABEL_SELECT | IM_KEYBOARD_SELECT;
```

```
      /* Request input from Reader Wedge */
      im_irl_v(IM_INFINITE_TIMEOUT,
               IM_ENABLE, IM_WEDGE_BEEP, IM_ENABLE,
               &source, input, &cc, &symbol);

      /* Display the input */
      printf("\n%s\n",input);

      /* Show a message if abort command in IRL was parsed */
      if (abort_flag)
      {
         printf("\nAbort program parsed.\n%d\n", abort_flag);

         /* Clear the abort flag */
         abort_flag = 0;
      }

   } while (input[0] != 'Q'  &&  input[0] != 'q' );

   /* Clear abort call back notification hook */
   im_clear_abort_callback();

   /* Switch the Reader Wedge back to virtual Wedge mode  */
   im_set_input_mode(IM_WEDGE);

}

/*
 ************************************************************************
* PURPOSE: callback function for abort program reader command detection
 *
 * NOTE: Need enable interrupt for this callback function,  because
 *        abort program is disable interrupt when it call callback function.
************************************************************************
 */

void _loadds Abort_Callback(IM_USHORT arg1, IM_USHORT arg2)
{
   /* Enable interrupt since the caller is disable the interrupt */
   _enable();
   /* Set the abort flag */
   abort_flag = arg1;

   /*********************************************************************/
   /* Note: In Borlandc "C" compiler, printf() routine works            */
   /*       In Microsoft "C" compiler, all routines referring __iob     */
   /*       variable won't work.  using routine (like putch()) or       */
   /*        directly output to console routine will work               */
   /*********************************************************************/
   printf("\nabort arg:%d, %d\n", arg1, arg2);  /* Does not work in Microsoft "C"*/
}
```

im_set_contrast

Purpose: This function sets the reader's LCD display contrast level. There are eight levels of contrast (0 to 7) from very light to very dark, which are the most frequently used contrast settings.

Syntax:

```
#include <im20lib.h>
    IM_STATUS im_set_contrast
    (IM_UCHAR display_contrast_level);
```

IN Parameters: The *display_contrast_level* parameter is one of these constants:

IM_MIN_CONTRAST	Level 0
IM_AVE_CONTRAST	Level 4
IM_MAX_CONTRAST	Level 7

OUT Parameters: None.

Return Value: This function returns one of the standard status codes defined in Appendix A, "Status Codes."

Notes: The functions im_set_contrast and im_get_contrast use a scale of 0 to 7 that is related to the scale of 0 to 31 used by im_increase_contrast and im_decrease_contrast. The following table compares the two scales.

This Value on Scale 0 to 7	Equates to This Value on Scale 0 to 31	Contrast Level
0	10	Very light
1	12	
2	14	
3	16	
4	18	
5	20	
6	22	
7	24	Very dark

This function has no effect on the JANUS 2050.

See Also: im_get_contrast, im_decrease_contrast, im_increase_contrast

Example

```
/******************* im_set_contrast ******************************/
/* Also with: im_get_contrast  */

#include <stdio.h>
#include <conio.h>
#include "im20lib.h"
#include "immsg.h"

void main (void)
{
IM_STATUS status;
IM_UCHAR  contrast;
IM_UCHAR  level;

   clrscr();

   for (level = 0; level < 10; level++)
   {

      status = im_set_contrast(level);

      /* If the most significant bit is set, there is an error */
      if (IM_ISERROR(status))
         printf("\nSet Contrast error = %x\n", status);
      else
         printf("\nContrast set to %d\n", level);

      status = im_get_contrast(&contrast);

      /* If the most significant bit is set, there is an error */
      if (IM_ISERROR(status))
         printf("\nGet Contrast error = %x\n", status);
      else
         printf("\nContrast is %d\n", level);

      /* Pause to see result */
      printf("\nAny key to continue");
      getch();
   }
}
```

im_set_control_key

Purpose: This function enables or disables the **Ctrl** key setting. When disabled, none of the key combinations that use the **Ctrl** key will work. For example, the warm boot key sequence **Ctrl-Alt-Del** will not work.

Syntax:

```
#include "im20lib.h"
IM_STATUS im_set_control_key
    (IM_CONTROL control_key);
```

IN Parameters: The *control_key* parameter is one of these constants:

IM_ENABLE	Enable the **Ctrl** key
IM_DISABLE	Disable the **Ctrl** key

OUT Parameters: None.

Return Value: This function returns one of the standard status codes defined in Appendix A, "Status Codes."

See Also: im_get_control_key

Example

```
See example for im_get_control_key.
```

im_set_display_mode

Purpose: This function sets the size mode, video mode, scroll mode, and character height of the display.

Syntax:

```
#include "im20lib.h"
IM_STATUS im_set_display_mode
    (IM_STD_SIZE_MODE size_mode,
    IM_STD_VIDEO_MODE video_mode,
    IM_SCROLL_MODE scroll_mode,
    IM_CHARACTER_HEIGHT char_ht);
```

IN Parameters: The *size_mode* parameter is required and is one of these constants:

IM_SIZE_MODE_80X25	80 x 25 text mode
IM_SIZE_MODE_20X16	20 x 16 text mode (2010 and 2020)
IM_SIZE_MODE_20X8	20 x 8 text mode (2010 and 2020)
IM_SIZE_MODE_10X16	10 x 16 text mode (2010 and 2020)
IM_SIZE_MODE_10X8	10 x 8 text mode (2010 and 2020)

If the *size_mode* parameter is set to IM_SIZE_MODE_80X25, you also need to set the video mode, scroll mode, and character height. If any other *size_mode* is set, the video mode, scroll mode, and character height are automatically set.

The *video_mode* parameter requires IM_SIZE_MODE_80X25. The standard PC BIOS supports up to video mode 13H. The reader only supports up to video mode 6. You can also set video modes 0 through 3 with IC.EXE.

The *video_mode* parameter is one of these constants:

IM_STD_VIDEO_MODE_0	40 x 25 (use for double-wide character)
IM_STD_VIDEO_MODE_1	40 x 25 (use for double-wide character)
IM_STD_VIDEO_MODE_2	80 x 25
IM_STD_VIDEO_MODE_3	80 x 25
IM_STD_VIDEO_MODE_4	300 x 200 2-color
IM_STD_VIDEO_MODE_5	300 x 200 monochrome
IM_STD_VIDEO_MODE_6	640 x 200 2-color (2050)

The *scroll_mode* requires IM_SIZE_MODE_80X25 and is one of these constants:

IM_LCD_SCROLL_AT_25	Scroll at line 25
IM_LCD_SCROLL_AT_16	Scroll at line 16 (2010 and 2020)
IM_LCD_SCROLL_AT_13	Scroll at line 13 (2050 and double height)
IM_LCD_SCROLL_AT_8	Scroll at line 8 (2010 and 2020)

The *char_ht* requires IM_SIZE_MODE_80X25 and is one of these constants:

IM_STANDARD_CHAR_HEIGHT	Standard height characters. You must have set the *scroll_mode* to scroll at line 25 or line 16.
IM_DOUBLE_CHAR_HEIGHT	Double height characters. You must have set the *scroll_mode* to scroll at line 8 or line 13.

OUT Parameters: None.

Return Value: This function returns one of the standard status codes defined in Appendix A, "Status Codes."

Notes: Changing display modes clears the screen.

The IRL input commands do not display correctly when the display is in 80 x 25 mode and the cursor has scrolled off the display. To alleviate this problem, set the display mode to one of the other modes, such as 20 x 16.

If your application uses an 8 x 10 or 16 x 10 (double width) display size and direct video addressing, the application will lock up because the display size does not provide enough characters to be detected by the hardware and displayed to the JANUS reader's screen. To avoid this problem, you must write the video using the BIOS when using a double width display size. For example, you can use the Borland global variable *directvideo*, which determines whether video writing goes to BIOS or direct video.

Note: *Intermec recommends that all applications write video using the BIOS instead of using direct video addressing.*

The mode does not change if there are conflicting parameters, such as trying to set the *video_mode* to 16 lines and the *scroll_mode* at line 8.

If you want to use one of the graphics modes, use the appropriate function call provided by your programming language. For example, Borland uses initgraph and Microsoft uses _setvideomode.

You must install the Reader Wedge to use this function. For information on loading RWTSR.EXE, see "Runtime Requirements" earlier in this chapter.

See Also: im_get_display_mode, im_set_follow_cursor

Example

```
See example for im_viewport_setxy.
```

im_set_follow_cursor

Purpose: When the display is in 80 x 25 mode, the viewport follows the cursor as it moves. This function enables or disables the follow-the-cursor feature.

Syntax:

```
#include "im20lib.h"
IM_STATUS im_set_follow_cursor
    (IM_CONTROL follow_cursor);
```

IN Parameters: The *follow_cursor* parameter is one of these constants:

IM_ENABLE	Enable follow-the-cursor mode
IM_DISABLE	Disable follow-the-cursor mode

OUT Parameters: None.

Return Value: This function returns one of the standard status codes defined in Appendix A, “Status Codes.”

Notes: The display must be in 80 x 25 mode for this function to work correctly.

This function has no effect on the JANUS 2050.

See Also: im_get_follow_cursor

Example

```
See example for im_get_follow_cursor.
```

im_set_input_mode

Purpose: This function clears the input buffers and sets the input manager to one of three modes: Wedge, Programmer, or Desktop.

Syntax:
```
#include <im20lib.h>
void im_set_input_mode (IM_MODE mode);
```

IN Parameters: The *mode* parameter is one of these constants:

IM_WEDGE	Wedge mode
IM_PROGRAMMER	Programmer mode
IM_DESKTOP	Desktop mode

OUT Parameters: None.

Return Value: None.

Notes: You must install the Reader Wedge to use this function. For information on loading RWTSR.EXE, see "Runtime Requirements" earlier in this chapter.

There are three different reader input modes: Wedge, Programmer, or Desktop. These modes affect how the reader interprets and stores input.

Wedge Mode Keypad and label inputs go into the keyboard buffer with minimal filtering. Wedge mode is the default mode at the DOS prompt after reader services (RSERVICE.EXE) is loaded. Use Wedge mode when the reader is running an application that uses the Virtual Wedge. When in this mode, use standard input functions to retrieve keypad or label input.

Wedge mode does not take full advantage of the reader's power management capabilities. You will probably notice reduced battery life when the reader is in this mode compared to either Programmer or Desktop mode. For more information on power management, see Chapter 4, "Advanced Programming."

Programmer Mode Inputs are echoed to the screen. Reader commands are executed and saved. Use Programmer mode when the reader is executing IRL commands. In general, when you are running an application that uses the function libraries or software interrupts, the reader should be in Programmer mode.

Desktop Mode The application is responsible for retrieving and displaying input. Use Desktop mode when you need detailed information about a pressed key. Each character returned consists of four bytes: the ASCII code, scan code, and two bytes for the keyboard flags (**Shift**, **Ctrl**, **Alt**). See the structure IM_KEYCODE in IM20LIB.H for further details.

For more information about reader input modes, see "Reader Input Modes" in Chapter 4, "Advanced Programming."

Upon exiting the program, set the input mode to IM_WEDGE so that DOS will work as expected.

See Also: im_get_input_mode, im_receive_input

Example

```
See example for im_get_input_mode.
```

im_set_keyclick

Purpose: Each time you press a key, the reader can emit a click. This function enables or disables the keyclick.

Syntax:

```
#include "im20lib.h"
IM_STATUS im_set_keyclick
    (IM_CONTROL keyclick_status);
```

IN Parameters: The *keyclick_status* parameter is one of these constants:

IM_ENABLE	Enable the keyclick
IM_DISABLE	Disable the keyclick

OUT Parameters: None.

Return Value: This function returns one of the standard status codes defined in Appendix A, “Status Codes.”

See Also: im_get_keyclick

Example

```
See example for im_get_keyclick.
```

im_set_viewport_lock

Purpose: This function enables or disables the keys that move the viewport.

Syntax:
```
#include "im20lib.h"
IM_STATUS im_set_viewport_lock
    (IM_CONTROL viewport_lock);
```

IN Parameters: The *viewport_lock* parameter is one of these constants:

IM_ENABLE	Lock the viewport
IM_DISABLE	Unlock the viewport

OUT Parameters: None.

Return Value: This function returns one of the standard status codes defined in Appendix A, "Status Codes."

Notes: The display must be in 80 x 25 mode for this function to work correctly.

This function controls whether the viewport moves (unlocked) or does not move (locked) when the cursor movement keys are pressed.

This function has no effect on the JANUS 2050.

See Also: im_get_viewport_lock, im_set_follow_cursor

Example

```
See example for im_get_viewport_lock.
```

im_set_warm_boot

Purpose: This function enables or disables the **Ctrl-Alt-Del** warm boot key sequence. When disabled, the **Ctrl-Alt-Del** key sequence does not reboot the reader.

Syntax:

```
#include "im20lib.h"
IM_STATUS im_set_warm_boot
    (IM_CONTROL warm_boot);
```

IN Parameters: The *warm_boot* parameter is one of these constants:

IM_ENABLE	Enable **Ctrl-Alt-Del** warm boot
IM_DISABLE	Disable **Ctrl-Alt-Del** warm boot

OUT Parameters: None.

Return Value: This function returns one of the standard status codes defined in Appendix A, “Status Codes.”

See Also: im_get_warm_boot

Example

```
See example for im_get_warm_boot.
```

im_setup_trx

Purpose: This function sets up the protocol parameters for a DCM transaction. If your program calls a DCM function without first calling im_setup_trx, then the default protocol values listed below are used.

For a detailed discussion of programming database transactions, see Chapter 5, "Using the PSK With DCM."

Syntax:

```
#include "im20lib.h"
IM_USHORT im_setup_trx
    (IM_COM_PORT port_id,
    IM_USHORT tx_timeout,
    IM_USHORT rx_timeout,
    IM_USHORT db_protocol,
    IM_UCHAR sys_delimiter,
    IM_UCHAR parse_char,
    IM_USHORT packet_max_len,
    int  hsafety);
```

IN Parameters: The *port_id* parameter identifies the communications port as follows:

IM_COM1	COM1 (Default)
IM_COM4	COM4 (RF only)

The *tx_timeout* parameter is the transmit timeout and is a numeric value in seconds or IM_INFINITE_TIMEOUT. The numeric range is between 0 and 65,000 seconds. The default is IM_INFINITE_TIMEOUT.

The *rx_timeout* parameter is the receive timeout and is a numeric value in seconds or IM_INFINITE_TIMEOUT. The numeric range is between 0 and 65,000 seconds. The default is IM_INFINITE_TIMEOUT.

The *db_protocol* parameter is one or more of these constants:

IM_DBPACE_ENABLE	Enable pacing for database read. The database sends one record to the reader and then waits for a request for the next record. Default is on.
IM_BATCH_ENABLE	Store all records in standby file, if one is enabled with *hsafety*. Default is off, no standby file.
IM_DOBEST_ENABLE	Ignore NAK and continue sending records when uploading standby records. Default is off: stop sending records on NAK.

The *sys_delimiter* parameter is the default transaction ID delimiter set in DCM. Default is a comma (,).

The *parse_char* parameter is the user-defined, data field-specific delimiter set in DCM. Default is a comma (,).

The *packet_max_len* parameter defines the maximum length for a transaction packet and is one of these values:

0	Use the default value and do not send a value.
-1	Use the previous setting.
1-256	Set the maximum length to this number. Default is 256.

The *hsafety* parameter is the file handler for the standby file. Valid values are:

0	No standby file. Default.
num	Use a standby file. A nonzero value that is the file handler created from `open( )`. See Notes below.

OUT Parameters: None.

Return Value: This function returns one of the status codes from the following table. For more information on these return codes, see Chapter 5, "Using the PSK With DCM."

IM_DCM_OK	1A00	Protocol parameters set successfully
IM_DCM_FILE_ERR	9A09	Standby file I/O error

Notes: To use a standby file, you must set *hSafety* by opening the file. The action of the standby file is determined by IM_BATCH_ENABLE.

If IM_BATCH_ENABLE is off, only transactions that cannot be sent are logged into the file. Usually this happens when the transmission times out.

If IM_BATCH_ENABLE is on, then all transactions are logged into the standby file. To transmit the standby file, you must set IM_BATCH_ENABLE to off, or use im_standard_trx with IM_ZIP_STANDBY to flush the standby records.

See Chapter 5, “Using the PSK With DCM.”

See Also: im_standard_trx, im_parse_host_response

Example

```
See the sample program in Chapter 5, “Using the PSK With DCM.”
```

im_sound

Purpose: This function generates a beep of specified pitch and duration. For example, use a soft beep for library use, a loud beep for manufacturing use, or a unique beep to distinguish between other readers.

Syntax:

```
#include <im20lib.h>
IM_STATUS im_sound
    (IM_USHORT pitch,
    IM_USHORT duration,
    IM_USHORT volume);
```

IN Parameters: The *pitch* parameter is the frequency of the beep you want the reader to make. The numeric range for *pitch* is from 20 to 20,000 Hz. You can also use one of these constants:

IM_HIGH_PITCH	2400 Hz
IM_LOW_PITCH	1200 Hz
IM_VERY_LOW_PITCH	600 Hz

The *duration* parameter is the length of the beep. The numeric range for *duration* is from 1 to 65,000 ms. You can also use one of these constants:

IM_BEEP_DURATION	50 ms
IM_CLICK_DURATION	4 ms

The *volume* parameter is one of these constants:

IM_OFF_VOLUME	Volume is off
IM_QUIET_VOLUME	Volume is quiet
IM_NORMAL_VOLUME	Volume is normal
IM_LOUD_VOLUME	Volume is loud
IM_EXTRA_LOUD_VOLUME	Volume is extra loud
IM_CURRENT_VOLUME	Volume is the current volume

OUT Parameters: None.

Return Value: This function returns one of the standard status codes defined in Appendix A, “Status Codes.”

Example

```
/******************* im_sound ************************************/
#include <time.h>
#include <conio.h>
#include <im20lib.h>

enum NOTES    /* Enumeration of notes and frequencies     */
{
   C0 = 262, D0 = 296, E0 = 330, F0 = 349, G0 = 392, A0 = 440, B0 = 494,
   C1 = 523, D1 = 587, E1 = 659, F1 = 698, G1 = 784, A1 = 880, B1 = 988,
   EIGHTH = 125, QUARTER = 250, HALF = 500, WHOLE = 1000, END = 0
} song[] =    /* Array initialized to notes of song     */
{
    C1, HALF, G0, HALF, A0, HALF, E0, HALF, F0, HALF, E0, QUARTER,
    D0, QUARTER, C0, WHOLE, END
};

/* Predefined the play volume */
  IM_USHORT volume[] = {
  IM_NORMAL_VOLUME,              /* Play normal volume     */
  IM_EXTRA_LOUD_VOLUME,          /* Play extra loud volume */
  IM_QUIET_VOLUME,               /* Play quiet volume      */
  IM_NORMAL_VOLUME               /* Play normal volume     */
};

void main (void)
{
int note;
int index;

   /* Play 4 times with 5 seconds dealy between each play */
   for ( index = 0; index < 4; index++)
   {

      for ( note = 0; song[note] != 0; note += 2 )
         im_sound( song[note], song[note + 1], volume[index]);

      /* Delay 5 seconds for next throughput*/
      im_standby_wait(5000);
   }
}
```

im_standard_trx

Purpose: This function packs reader data into a DCM transaction. The function packs the data, transmits the packed transaction through a communications port, and then reads the host response message from the communications port.

For a detailed discussion of programming database transactions, see Chapter 5, "Using the PSK With DCM."

Syntax:

```
#include "im20lib.h"
IM_USHORT im_standard_trx
    (IM_UCHAR *trans_id,
    IM_UCHAR request_type,
    IM_UCHAR *send_var_array[],
    IM_USHORT max_str_len,
    IM_UCHAR *ret_string,
    IM_DCMSTRN ret_array[],
    IM_UCHAR *oper_type);
```

IN Parameters: The **trans_id* parameter is a pointer to the transaction ID string. The transaction ID must match the ID already defined in DCM. See the example in Chapter 5, "Using the PSK With DCM."

The *request_type* parameter is associated with the transaction ID and either accesses the remote database or manipulates the standby file. The *request_type* is one of these constants:

For Accessing the Remote Database

IM_DBRQT_DATA	D	Request data
IM_DBRQT_NEXTROW	N	Request next row
IM_DBRQT_DISCARD	S	Delete the following records

For Manipulating the Standby File

IM_ERROR_RECORD	E	Report error record without advancing the record pointer in the standby file
IM_ADVANCE_RECORD	A	Report error record and advance the record pointer in the standby file
IM_ZIP_STANDBY	Z	Flush the standby records

The **send_var_array[]* parameter is valid only when *request_type* is "D." It is a pointer to the array containing the data to send to the database. Each element in the array corresponds to a DCM transaction field.

The *max_str_len* parameter is the maximum length of the return string buffer.

OUT Parameters: The **ret_string* parameter is a pointer to the return string buffer. The buffer contains the database server's transaction message unless a timeout occurred.

If the reader receives IM_DCM_TIMEOUT, then the communication error status is placed in the first two bytes of the return buffer. You can check this communication status code against the list in Appendix A, "Status Codes."

The return string buffer must be large enough to hold a complete packet from the controller.

The *ret_array[]* parameter is the array containing pointers to the variables that hold the data returned from the database. The received data field is parsed and then placed into the variables.

The **oper_type* parameter is a pointer to the operation type returned by the remote database and is one of these constants:

IM_DB_READ	R	Read records from the database
IM_DB_INSERT	W	Write records to the database
IM_DB_UPDATE	U	Update records in the database
IM_DB_DELETE	D	Delete records from the database
IM_DB_UNKNOWN	X	Unknown

Return Value: This function returns one of the status codes from the following table. For more information on these return codes, see Chapter 5, "Using the PSK With DCM."

Return Code	Hex	Description
IM_DCM_OK	1A00	Transaction successful Retrieved record from standby file All standby records sent successfully
IM_DCM_ACK_RCD	1A01	Transaction successful, ACK
IM_DCM_MSG_RCD	1A10	Received message on Read operation
IM_DCM_NAK_RCD	5A02	Transaction failed, NAK received
IM_DCM_NO_RECORD	5A07	Standby file is empty
IM_DCM_TRUNCATED	5A11	Data was truncated
IM_DCM_EXCESSFIELD	5A12	Too many fields
IM_DCM_STANDBY	5A20	Record stored in standby file successfully
IM_DCM_STANDBY_TXERR	5A21	Record stored in standby file, but upload standby error
IM_DCM_NOTRANSMIT	9A03	Transmit failed Transmit channel error
IM_DCM_TIMEOUT	9A04	Receive port timeout
IM_DCM_HEAD_ERR	9A05	Packet header error
IM_DCM_TRANSID_ERR	9A06	Missing transaction ID
IM_DCM_FILE_ERR	9A08	Standby file I/O error
IM_DCM_NO_HANDLE	9A09	Missing standby file
IM_DCM_FILLED	9A40	Buffer overflow
IM_DCM_FILLED_TXERR	9A41	Buffer overflow with upload standby error

Notes: You can use the **oper_type* and *request_type* to verify that the database, the DCM interface, and the JANUS transaction are configured properly. For example, if the *request_type* is IM_DBRQT_DATA (read or write), but the **oper_type* is IM_DB_DELETE (delete), then you need to correct the configuration.

See Chapter 5, “Using the PSK With DCM.”

See Also: im_setup_trx, im_parse_host_response

Example

```
See the sample program in Chapter 5, “Using the PSK With DCM.”
```

im_standby_wait

Purpose: This function requests that reader services wait in standby mode for a specific period of time. You use this function to suspend the reader for a length of time to save the battery power.

Syntax:

```
#include <im20lib.h>
IM_STATUS im_standby_wait
    (IM_UINT timeout);
```

IN Parameters: The *timeout* parameter is the amount of time to wait in standby mode. You can exit the function before data is received or before the timeout occurs by performing an application break sequence (see your JANUS user's manual). The return status indicates whether the function was successful, a timeout occurred, or the application break was received.

The *timeout* parameter is a number or one of these constants:

1 to 65,534 ms	Numeric range
IM_ZERO_TIMEOUT	No wait
IM_INFINITE_TIMEOUT	Wait forever

OUT Parameters: None.

Return Value: This function returns one of the standard status codes defined in Appendix A, "Status Codes."

Notes: You must install the Reader Wedge to use this function. For information on loading RWTSR.EXE, see "Runtime Requirements" earlier in this chapter.

See Also: None.

Example

```
See example for im_irl_a.
```

im_transmit_buffer

Purpose: This function transmits the contents of a data buffer through a designated communications port. This function continues operating until the buffer transmission is complete or until an error status is detected.

Syntax:

```
#include <im20lib.h>
IM_STATUS im_transmit_buffer
    (IM_COM_PORT port_id,
    IM_USHORT user_length,
    IM_UCHAR far *data_buffer,
    IM_USHORT timeout);
```

IN Parameters: The *port_id* parameter identifies the communications port as follows:

IM_COM1	COM1
IM_COM2	COM2, scanner port (2010 and 2050)
IM_COM4	COM4 (RF only)

The *user_length* parameter is the length of the data string that you want to transmit.

The *data_buffer* parameter is a far pointer to the data array that you want to transmit.

The *timeout* parameter is the transmission timeout period. You can exit the function before data is sent or before the timeout occurs by performing an application break sequence (see your JANUS user's manual). The return status indicates whether the function was successful, a timeout occurred, or the application break was received.

The *timeout* parameter is a number or one of these constants:

1 to 65,534 ms	Numeric range
IM_ZERO_TIMEOUT	No wait
IM_INFINITE_TIMEOUT	Wait forever

OUT Parameters: None.

Return Value: This function returns one of the standard status codes defined in Appendix A, "Status Codes."

Notes: You must install a protocol handler to use this function.

See Also: im_receive_buffer, im_transmit_buffer_no_wait, im_transmit_buffer_noprot

---

Example

```
/******************** im_transmit_buffer ******************************/
/* Also with: im_cancel_tx_buffer  */
/* Note: Must have reader services and comm protocol handler installed to  */
/*       enable send/receive buffer functions.                             */
/*       Run these TSR's at DOS prompt before running these tests:         */
/*        rservice                                                         */
/*        phimec 1  ( 1 is COM1)                                           */

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#include "im20lib.h"
#include "immsg.h"

static IM_COM_PORT  com_port = IM_COM1;    /* Uses COM1*/

static IM_UCHAR     *data_str[] = {
   "Hi There, It must have three lines down the road from this demo",
   "The line 2 demonstration proves the transmit buffer is working",
   "The demonstration of the transmit buffer is Done",
   NULL
};

void main (void)
{
int          index;
size_t       actual_count;
IM_STATUS    status;
IM_UCHAR     data_buffer[160];

   /* Call im_cancel_tx_buffer to initialize the communications port and its buffer */
   status = im_cancel_tx_buffer(com_port);
   printf("Cancel Tx status: %xH\n", status);

   /* Sending data string loop until it reaches NULL */
   for (index =0; data_str[index] != NULL; index++)
   {
      /* Copy string to big buffer */
      strcpy(data_buffer, data_str[index]);
      actual_count = strlen(data_buffer);     /* Get its length */

      /* Call Intermec function, timeout within 50000 milliseconds */
      status = im_transmit_buffer(com_port, actual_count, data_buffer, 50000);

      if (IM_ISSUCCESS(status))
         printf("Line %d Sent\n", index + 1);
      else
         printf("Transmit error\nStatus = %xH\n", status);
   }
   printf("Transmit Done\n");
}
```

im_transmit_buffer_no_wait

Purpose: This function transmits the contents of a buffer and passes control back to its client.

Syntax:

```
#include <im20lib.h>
IM_STATUS im_transmit_buffer_no_wait
    (IM_COM_PORT port_id,
    IM_COM_DATA_BUFFER far *data_struct);
```

IN Parameters: The *port_id* parameter identifies the communications port as follows:

IM_COM1	COM1
IM_COM2	COM2, scanner port (2010 and 2050)
IM_COM4	COM4 (RF only)

The *data_struct* parameter is a far pointer to the data array that you want to transmit.

OUT Parameters: None.

Return Value: This function returns one of the standard status codes defined in Appendix A, “Status Codes.”

Notes: You must install a protocol handler to use this function.

This function differs from im_transmit_buffer in that the client must set the buffer structure and monitor the buffer status until transmission is complete.

Use this function for multiple buffer transmissions in conjunction with checking the status in *data_struct*.

You must build and maintain a *data_struct* of the type IM_COM_DATA_BUFFER as described in IM20LIB.H.

See Also: im_transmit_buffer_noprot, im_cancel_tx_buffer

Example

```
/******************* im_transmit_buffer_no_wait ***********************/
/* Also with: im_cancel_tx_buffer  */
/* Note: Must have reader services and comm protocol handler installed to  */
/*       enable send/receive buffer functions.                             */
/*       Run these TSR's at DOS prompt before running these tests:         */
/*        rservice                                                         */
/*        phimec 1  ( 1 is COM1)                                           */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "im20lib.h"
#include "immsg.h"

static IM_COM_PORT  com_port = IM_COM1;    /* Uses COM1*/

static IM_UCHAR     *data_str[] = {
   "Hi There, It must have four lines down the road from this demo\r"
   "The line 2 demonstration proves the Transmit Buffer No Wait working\r",
   "The line 3 demonstration proves the Transmit Buffer No Wait working\r",
   "The demonstration of the Transmit Buffer No Wait is Done\r",
   NULL
};

void main (void)
{
IM_USHORT           index;
IM_COM_DATA_BUFFER  com_buff_struct; /*data buffer structure for receive/transmit */
IM_STATUS           status;
IM_UCHAR            data_buff[300];

   /* Get the transmit environment ready */
   status = im_cancel_tx_buffer(com_port);
   printf("Cancel Tx status: %xH\n", status);

   /* Transmitting data string loop until it reaches NULL */
   for (index =0; data_str[index] != NULL; index++)
   {
      /* Copy string to transmit into big buffer */
      strcpy(data_buff, data_str[index]);

      /* Set up comm buffer parameters */
      com_buff_struct.command       = IM_CU_TRANSMIT;
      com_buff_struct.user_length   = strlen(data_buff);
      com_buff_struct.comm_length   = 0;
      com_buff_struct.protocol_mode = IM_NO_PROTOCOL;
      com_buff_struct.eom_char      = 0;
      com_buff_struct.data_ptr      = (char far *)data_buff;
      com_buff_struct.protocol_xfer_status = IM_COMM_INUSE;

      /* Call Intermec routine */
      status = im_transmit_buffer_no_wait(com_port, (IM_COM_DATA_BUFFER far *)
                                                    &com_buff_struct);

      if (IM_ISSUCCESS(status))
      {
         /* Wait for data transfer to complete */
         while (com_buff_struct.protocol_xfer_status == IM_COMM_INUSE)
            ;           /* Waiting loop, end while */
```

```
            printf("Line %d Sent\n", index + 1);
         }
         else
         {
            printf("Tx Error = %XH\n", status);
            printf("Buffer not sent\n");
         }

      }  /* for */

      printf ("Transmit Done\n");
}
```

im_transmit_buffer_noprot

Purpose: This function transmits a buffer without protocol and sends only the specified number of characters.

Syntax:

```
#include <im20lib.h>
IM_STATUS im_transmit_buffer_noprot
    (IM_COM_PORT port_id,
    IM_USHORT user_length,
    IM_UCHAR far *data_buffer
    IM_USHORT timeout);
```

IN Parameters: The *port_id* parameter identifies the communications port as follows:

IM_COM1	COM1
IM_COM2	COM2, scanner port (2010 and 2050)
IM_COM4	COM4 (RF only)

The *user_length* parameter is the maximum number of bytes to transmit.

The *data_buffer* parameter is a far pointer to the data array that you want to transmit.

The *timeout* parameter is the transmission timeout period. You can exit the function before data is sent or before the timeout occurs by performing an application break sequence (see your JANUS user's manual). The return status indicates whether the function was successful, a timeout occurred, or the application break was received.

The *timeout* parameter is a number or one of these constants:

1 to 65,534 ms	Numeric range
IM_ZERO_TIMEOUT	No wait
IM_INFINITE_TIMEOUT	Wait forever

If IM_INFINITE_TIMEOUT is selected, the function will not return until the end of message character has been received.

OUT Parameters: None.

Return Value: This function returns one of the standard status codes defined in Appendix A, "Status Codes."

Notes: You must install a protocol handler other than PHPCSTD.EXE to use this function.

If an existing transmit client is linked to the host, you must issue a cancel command first.

See Also: im_transmit_buffer_no_wait, im_cancel_tx_buffer, im_transmit_buffer

---

Example

```
/******************* im_transmit_buffer_noprot ***********************/
/* Also with: im_cancel_tx_buffer  */
/* Note: Must have reader services and comm protocol handler installed to  */
/*       enable send/receive buffer functions.                             */
/*       Run these TSR's at DOS prompt before running these tests:         */
/*        rservice                                                         */
/*        phimec 1  ( 1 is COM1)                                           */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "im20lib.h"
#include "immsg.h"

static IM_COM_PORT  com_port = IM_COM1;    /* Uses COM1*/

static IM_UCHAR     *data_str[] = {
   "Hi There, It must have three lines down the road from this demo",
   "The line 2 demo proves the Transmit Buffer No Protocol is working",
   "The demonstration of the Transmit Buffer No Protocol is Done",
   NULL
};

void main (void)
{
IM_STATUS status;
IM_USHORT index;
IM_UCHAR  data_buffer[300];

   /* PHIMEC protocol handler must be installed to run this routine */
   /* It only works with PHIMEC protocol                            */

   /* Get the transmit environment ready */
   status = im_cancel_tx_buffer(com_port);
   printf("Cancel Tx status: %xH\n", status);

/* Sending data string loop until it reaches NULL */
   for (index =0; data_str[index] != NULL; index++)
   {
      /* Copy string to transmit into big buffer */
      strcpy(data_buffer, data_str[index]);

      /* Call Intermec routine with infinite timeout */
      status = im_transmit_buffer_noprot(com_port, strlen(data_buffer),
                                         data_buffer, IM_INFINITE_TIMEOUT);
```

```
      if (IM_ISSUCCESS(status))
         printf("Line %d Sent\n", index + 1);
      else
         printf("Tx Error\nStatus = %xH\n", status);
   }

   printf("Tx No Protocol Done\n");
}
```

im_transmit_byte

Purpose: This function transmits one byte of data out from the designated communications port. This function is identical to MS-DOS INT 14H Service 01H.

Syntax:

```
#include <im20lib.h>
IM_STATUS im_transmit_byte
    (IM_COM_PORT port_id,
    IM_UCHAR transmit_byte);
```

IN Parameters: The *port_id* parameter identifies the communications port as follows:

IM_COM1	COM1
IM_COM2	COM2, scanner port (2010 and 2050)
IM_COM4	COM4 (RF only)

The *transmit_byte* parameter is the byte of data to transmit.

OUT Parameters: None.

Return Value: This function returns one of the standard status codes defined in Appendix A, “Status Codes.”

Notes: This function requires the PC standard protocol handler PHPCSTD.EXE.

Intermec recommends using im_transmit_buffer (with a *user_length* of one) instead of using im_transmit_byte.

You can use this function for an acknowledgment.

See Also: im_receive_byte, im_transmit_buffer

Example

```
/******************* im_transmit_byte  ***********************************/
/* Uses Borland _bio_serialcom or Microsoft _bios_serialcom  to init the   */
/* com port for single byte.                                               */
/* Operations. It sets up for 9600,E,7,1 on serial port                    */
/* Note: Phimec protocol handler must NOT be installed to run this routine */

#include <stdio.h>
#include <dos.h>
#include <conio.h>
#include <stdlib.h>
#include <bios.h>
#include <string.h>
#include "im20lib.h"
#include "immsg.h"

/* Byte oriented defines */
#define COM1       0
#define SETTINGS (_COM_9600 | _COM_CHR7 | _COM_STOP1 | _COM_EVENPARITY )

static IM_COM_PORT  com_port = IM_COM1;      /* Uses COM1*/

void main (void)
{
IM_UCHAR  outchar;
IM_STATUS status;

   printf("\nChoose a protocol:\n");
   printf("1) PC Standard\n");
   printf("2) No protocol\n");

   if (getche() == '2')       /* Call bios to initialize com port */
   {
      /* Communications port init */
      _bios_serialcom(_COM_INIT, COM1, SETTINGS);
   }

   /* Display instruction */
   printf ("Enter characters\n");
   printf (" to be transmitted\n");
   printf (" <ESC> to quit\n");

   /* Phimec protocol handler must NOT be installed ...*/
   while ((outchar = getche()) != '\x1B')     /* Send char until <ESC> char*/
   {
      /* Call Intermec function */
      status = im_transmit_byte( com_port, outchar);
      if ( IM_ISERROR(status))
         im_message(status);
   }
}
```

im_unlink_comm

Purpose: This function removes the link between the Reader Wedge function and a designated communications port. After this function excutes, the Reader Wedge is not notified of receive buffer functions.

You only need to use this procedure one time, at the end of your program.

Syntax:

```
#include <im20lib.h>
IM_STATUS im_unlink_comm
    (IM_COM_PORT port_id);
```

IN Parameters: The *port_id* parameter identifies the communications port as follows:

IM_COM1	COM1
IM_COM2	COM2, scanner port (2010 and 2050)
IM_COM4	COM4 (RF only)

OUT Parameters: None.

Return Value: This function returns one of the standard status codes defined in Appendix A, “Status Codes.”

Notes: You must install a protocol handler to use this function.

You must install the Reader Wedge to use this function. For information on loading RWTSR.EXE, see “Runtime Requirements” earlier in this chapter.

See Also: im_link_comm

Example

```
See example for im_receive_input.
```

im_viewport_end

Purpose: This function sets the viewport to the lower right corner (end) of the virtual display.

Syntax:
```
#include <im20lib.h>
void im_viewport_end(void);
```

IN Parameters: None.

OUT Parameters: None.

Return Value: None.

Notes: This function has no effect on the JANUS 2050.

See Also: im_cursor_to_viewport, im_viewport_home, im_viewport_move, im_viewport_page_down, im_viewport_page_up, im_viewport_to_cursor

Example

```
See example for im_viewport_setxy.
```

im_viewport_getxy

Purpose: This function retrieves the column and row of the upper left corner of the viewport.

Syntax:

```
#include "im20lib.h"
IM_STATUS im_viewport_getxy
    (IM_USHORT *row,
    IM_USHORT *col);
```

IN Parameters: None.

OUT Parameters: The *row* parameter returns the row value (Y-coordinate) of the viewport.

The *col* parameter returns the column value (X-coordinate) of the viewport.

Return Value: This function returns one of the standard status codes defined in Appendix A, "Status Codes."

Notes: This function is valid only when the display is in 80 x 25 mode.

This function has no effect on the JANUS 2050.

See Also: im_viewport_end, im_viewport_home, im_viewport_move, im_viewport_page_up, im_viewport_page_down, im_viewport_setxy, im_viewport_to_cursor

Example

```
See example for im_viewport_setxy.
```

im_viewport_home

Purpose: This function sets the viewport to the upper left corner (home) of the virtual display.

Syntax:
```
#include <im20lib.h>
void im_viewport_home(void);
```

IN Parameters: None.

OUT Parameters: None.

Return Value: None.

Notes: This function has no effect on the JANUS 2050.

See Also: im_cursor_to_viewport, im_viewport_end, im_viewport_move, im_viewport_page_down, im_viewport_page_up, im_viewport_to_cursor

Example

```
See example for im_viewport_setxy.
```

im_viewport_move

Purpose: This function moves the viewport the specified distance and direction.

Syntax:

```
#include <im20lib.h>
IM_STATUS im_viewport_move
    (IM_VIEWPORT_DIRECTION direction,
    IM_USHORT distance,
    IM_USHORT *row,
    IM_USHORT *col);
```

IN Parameters: The *direction* parameter is one of these constants:

IM_VIEWPORT_LEFT	Move left
IM_VIEWPORT_RIGHT	Move right
IM_VIEWPORT_UP	Move up
IM_VIEWPORT_DOWN	Move down

The *distance* parameter indicates the number of units to move the viewport and is either a numeric value between 1 and 70 or is IM_DEFAULT_DISTANCE.

If the distance parameter is IM_DEFAULT_DISTANCE, the default step size is used.

The distance moved is a configurable parameter. For more information on configuring the viewport, see your JANUS user's manual.

OUT Parameters: The *row* parameter returns the row number to which the viewport has moved.

The *col* parameter returns the column number to which the viewport has moved.

Return Value: This function returns one of the standard status codes defined in Appendix A, "Status Codes."

Notes: The (*row, column*) pair represents the upper left corner of the viewport being displayed.

The minimum value for both the *row* and *column* is (0,0), which is the upper left corner of the virtual window. The maximum value is determined by the video mode and the number of lines and columns displayed.

For example, for the normal video mode 3 (80 x 25 display) with a 20 x 16 display size, the maximum value of *row* is 9 (25 minus 16) and the maximum value of *column* is 60 (80 minus 20).

The row and column viewport settings are set to the maximum allowed for the current mode and display size. To determine the actual values set, check the returned values of *row* and *col*.

This function has no effect on the JANUS 2050.

See Also: im_viewport_end, im_viewport_home, im_viewport_page_down, im_viewport_page_up, im_viewport_to_cursor, im_cursor_to_viewport

---

Example

```
See example for im_viewport_setxy.
```

im_viewport_page_down

Purpose: This function moves the viewport down one viewport length.

Syntax:
```
#include <im20lib.h>
void im_viewport_page_down(void);
```

IN Parameters: None.

OUT Parameters: None.

Return Value: None.

Notes: This function has no effect on the JANUS 2050.

See Also: im_cursor_to_viewport, im_viewport_end, im_viewport_home, im_viewport_move, im_viewport_page_up, im_viewport_to_cursor

Example

```
See example for im_viewport_setxy.
```

im_viewport_page_up

Purpose: This function moves the viewport up one viewport length.

Syntax:
```
#include <im20lib.h>
void im_viewport_page_up(void);
```

IN Parameters: None.

OUT Parameters: None.

Return Value: None.

Notes: This function has no effect on the JANUS 2050.

See Also: im_cursor_to_viewport, im_viewport_end, im_viewport_home, im_viewport_move, im_viewport_page_down, im_viewport_to_cursor

Example

```
See example for im_viewport_setxy.
```

im_viewport_setxy

Purpose: This function sets the viewport row and column to a specific value when moving between two screens.

Syntax:

```
#include <im20lib.h>
IM_STATUS im_viewport_setxy
    (IM_USHORT *row,
    IM_USHORT *col);
```

IN/OUT Parameters: The *row* parameter sets the row value (Y-coordinate) of the viewport.

The *col* parameter sets the column value (X-coordinate) of the viewport.

Return Value: This function returns one of the standard status codes defined in Appendix A, “Status Codes.”

Notes: The (*row*, *col*) pair represents the upper left corner of the viewport being displayed. The minimum values for both the *row* and *col* are (0,0), which is the upper left corner of the virtual window. For the normal video mode 3 (80 x 25 display) with 20 x 16 display character size, the maximum value of *row* is 9 (25 minus 16) and the maximum value of *col* is 60 (80 minus 20).

The row and column viewport settings are set to the maximum allowed for the current mode and display size. To determine the actual values set, check the returned values of *row* and *col*.

This function has no effect on the JANUS 2050.

See Also: im_viewport_move

Example

```
/******************* im_viewport_setxy ***********************************/
/* Also with: im_viewport_getxy */
/*   im_viewport_home           */
/*   im_viewport_end            */
/*   im_viewport_page_down      */
/*   im_viewport_page_up        */
/*   im_viewport_move           */
/*   im_set_display_mode        */
/*   im_get_display_mode        */
/*   im_viewport_to_cursor      */
/*   im_cursor_to_viewport      */

#include <stdio.h>
#include <dos.h>
#include <conio.h>
#include <stdlib.h>
#include "im20lib.h"

#define  SOUND_TIME   200           /* 200 milliseconds */

void main (void)
{
IM_USHORT              row,          col;
IM_STD_SIZE_MODE       std_size,     cmode_size;
IM_STD_VIDEO_MODE      std_video,    cmode_video;
IM_SCROLL_MODE         std_scroll,   cmode_scroll;
IM_CHARACTER_HEIGHT    std_charhigh, cmode_charhigh;

/* Set display mode to 80 x 25 and remaining parameters standard. */
   std_size     = IM_SIZE_MODE_80X25;
   std_video    = IM_STD_VIDEO_MODE_3;
   std_scroll   = IM_LCD_SCROLL_AT_25;
   std_charhigh = IM_STANDARD_CHAR_HEIGHT;
   im_set_display_mode(std_size, std_video, std_scroll, std_charhigh);

   /* Set up mark for viewport operation fuctions */
   /* DOS Screen uses (1,1) as home but im functions start at (0,0) */
   gotoxy(  1,  1  );  cputs("1-viewport_home");      /* viewport_home mark*/
   gotoxy(  1,  1+1);  cputs("  row:0, col:0");
   gotoxy( 61, 24  );  cputs("2-viewport_end");       /* viewport_end mark*/
   gotoxy( 61, 24+1);  cputs("  row:23, col:60");
   gotoxy( 61,  1  );  cputs("3-viewport_page_up");   /* viewport_page_up */
   gotoxy( 61,  1+1);  cputs("  row:0, col:60");
   gotoxy( 21,  1  );  cputs("4-viewport_setxy");     /* viewport_setxy */
   gotoxy( 21,  1+1);  cputs("  row:0, col:20");
   gotoxy( 41,  1  );  cputs("5-viewport_move");      /* viewport_move */
   gotoxy( 41,  1+1);  cputs("  row:0, col:40");
   gotoxy( 41, 19  );  cputs("6-viewport_page_down"); /* vewport_page_down */
   gotoxy( 41, 19+1);  cputs("  row:18, col:40");
   gotoxy( 21, 19  );  cputs("7-viewport_to_cursor"); /* viewport_to_cursor*/
   gotoxy( 21, 19+1);  cputs("   row:18, col:20");

   /* Bring cursor back so that viewport will stay with it */
   gotoxy(  1,  1);

   /* Action 1: Set viewport back to Home */
   im_viewport_home();
   im_sound(IM_LOW_PITCH,SOUND_TIME,IM_LOUD_VOLUME);
   getch();
```

```
/* Action 2: Viewport End */
im_viewport_end();
im_sound(IM_LOW_PITCH,SOUND_TIME,IM_LOUD_VOLUME);
getch();

/* Action 3: Viewport Page Up */
im_viewport_page_up();
im_sound(IM_LOW_PITCH,SOUND_TIME,IM_LOUD_VOLUME);
getch();

/* Action 4: Viewport Setxy */
row = 0;  col = 20;
im_viewport_setxy(&row, &col);
im_sound(IM_LOW_PITCH,SOUND_TIME,IM_LOUD_VOLUME);
getch();

/* Action 5: Viewport Move */
im_viewport_move(IM_VIEWPORT_RIGHT, 20, &row, &col);
im_sound(IM_LOW_PITCH,SOUND_TIME,IM_LOUD_VOLUME);
getch();

/* Action 6: Viewport Page Down */
im_viewport_page_down();
im_sound(IM_LOW_PITCH,SOUND_TIME,IM_LOUD_VOLUME);
getch();

/* Action 7: set cursor at row:18, column:10 and viewport_to_cursor */
/*           Note: Cursor at the centor of viewport                 */
gotoxy(20+10, 18);          /* Need offset 10 to let cursor at center*/
im_viewport_to_cursor();
im_sound(IM_LOW_PITCH,SOUND_TIME,IM_LOUD_VOLUME);
getch();

/* Action 8: Viewport Getxy */
im_viewport_end();                        /* Force viewport to end */
/* In 80X25 mode, viewport_end causes the viewport move at (9,60) */
/* You should see value is row:9, col:60 */
im_viewport_getxy(&row, &col);
im_viewport_home();                       /* Force viewport to home */
gotoxy(1, 1  ); cputs  ("8-viewport_getxy");  /* Start at home again */
gotoxy(1, 1+1); cputs ("  Row:%d, Col:%d", row, col);
im_sound(IM_LOW_PITCH,SOUND_TIME,IM_LOUD_VOLUME);
getch();

/* Action 9: Mark ref. c->v  */
clrscr();
gotoxy(51, 10 );  cputs("Ref. C->V"); /* viewport_to_cursor*/
gotoxy( 1, 1);    cputs("9-C->V");
gotoxy( 1, 1+1);  cputs("Cursor at Ref.");
getch();
/* Action 9: Move viewport to end, then Do cursor_to_viewport */
row = 2;  col = 40;
im_viewport_setxy(&row, &col);
im_viewport_end();              /* Force viewport to end */
im_cursor_to_viewport();        /* Cursor go to center */
im_sound(IM_LOW_PITCH,SOUND_TIME,IM_LOUD_VOLUME);
getch();
/* Action 10: Get Display Mode */
im_get_display_mode(&cmode_size, &cmode_video, &cmode_scroll,
                    &cmode_charhigh);
if ( (std_size == cmode_size) && (std_video == cmode_video) &&
     (std_scroll == cmode_scroll) && (std_charhigh == cmode_charhigh) )
```

```
      printf("\nDisplay mode setup ok\n");
else
      printf("\nError in display_mode\nsetup\n");
   im_sound(IM_LOW_PITCH,SOUND_TIME,IM_LOUD_VOLUME);
   getch();
}
```

im_viewport_to_cursor

Purpose: This function centers the viewport around the cursor.

Syntax:
```
#include <im20lib.h>
void im_viewport_to_cursor(void);
```

IN Parameters: None.

OUT Parameters: None.

Return Value: None.

Notes: When the cursor is not displayed, use this function to move the viewport to the cursor.

This function has no effect on the JANUS 2050.

See Also: im_cursor_to_viewport, im_viewport_end, im_viewport_home, im_viewport_move, im_viewport_page_down

Example

```
See example for im_viewport_setxy.
```

3

Software Interrupts

This chapter explains how to use software interrupts in your programs.

Using the Software Interrupts

If PSK library functions are not available for your programming language, you can use software interrupts to invoke the special functions of the reader. These special functions are called *reader services.*

Reader services include several functions, ranging in complexity from controlling the beeper to controlling the internal power management software.

The reader supports other standard PC / AT interrupts that control the DOS and BIOS functions of a normal PC. It is beyond the scope of this manual to list all the normal PC interrupts. This chapter deals only with the interrupts specific to the JANUS family of PC-compatible readers.

Use the following tables as a quick reference point for finding the interrupts you need.

Interrupt Range	Usage
00H – 1FH	Internal hardware interrupts
20H – 3FH	DOS interrupts
3FH and above	Intermec-specific interrupts

Vector Interrupts

Vector	Standard PC	Reader
0	Divide by Zero	Divide by Zero
1	Single Step	Single Step
2	Nonmaskable Interrupt	Nonmaskable Interrupt
3	Breakpoint	Breakpoint
4	Overflow	Overflow
5	Print Screen or Bounds Check	Print Screen or Bounds Check
6	Invalid Opcode	Invalid Opcode
7	Coprocessor Not Available	Coprocessor Not Available
8	System Timer (IRQ 0) or Double Fault	System Timer (IRQ 0) or Double Fault
9	Keyboard (IRQ 1) or Coprocessor	Keyboard (IRQ 1) or Coprocessor Segment Overflow
A	Reserved	Slave PIC (IRQ 2) or Invalid TSS
B	COM	COM 2 or 4 (IRQ 3) or Segment Not Present COM 2 is available only on the 2010 or 2050
C	COM	COM 1 or 3 (IRQ 4) or Stack Exception
D	Hard Disk or LPT	Intermec Count Gathering (IRQ 5) or General Protection
E	Floppy Disk	Reserved (IRQ 6) or Page Fault
F	LPT	Reserved (IRQ 7)
10–1F	BIOS	BIOS
20–3F	DOS	DOS
40–43	BIOS	Pointers to BIOS Table
44–45	BIOS	Reserved
46	BIOS	Pointers to BIOS Table
47–49		Reserved
4A		User Alarm (RTC)
4B–59		Reserved
5A–5B	NET	Cluster Adapter
5C	NET	Net BIOS
5D–5F		Power Management (IPM)

Vector Interrupts (continued)

Vector	Standard PC	Reader
60–66		User Reserved
67	EMS Driver	EMS Driver
68–6F	VGA	Reserved
70	IRQ 8	RTC (IRQ 8)
71	IRQ 9	Reserved (IRQ 9)
72	IRQ 10	Keypad Scanner (IRQ 10)
73	IRQ 11	Reserved (IRQ 11)
74	IRQ 12	Reserved (IRQ 12)
75	IRQ 13	Reserved (IRQ 13)
76	IRQ 14	Reserved (IRQ 14)
77	IRQ 15	Power Management (IRQ 15)
78		System Configuration and Control Services
79		Display Services
7A		Bootstrap Services
7B		Reserved
7C		Beep Utility
7D		Reader Services
7E		Special Keypad Services
7F		Bootstrap Hook
80–F0	BASIC	BASIC
F1–FF	User Reserved	User Reserved

Programming With Software Interrupts

To invoke the Intermec utilities, you must have a programming language that supports interrupt-level routines. Most programming languages include this type of functionality, but the support methods vary from language to language.

For example, Microsoft C uses the INT86 and INT86X functions to execute software interrupts, and Microsoft QuickBasic uses the INTERRUPT and INTERRUPTX calls. For more information about interrupt functions, see the documentation for your compiler and the programming examples in Appendix B, "Sample Interrupt Programs."

Specific Software Interrupt Definitions

The following pages list software interrupts that are unique to the JANUS readers, along with notes on usage and examples of how each interrupt is generated in a program.

Note: *Do* ***not*** *run programs that use* ***PSK library functions*** *on your PC. If you attempt to run these programs, you will receive an error message and the program will not run.*

Caution
Do not run programs that use **Intermec-specific interrupt extensions** ***on your PC. If you attempt to run these programs, they will cause your PC to lock up and possibly corrupt your system BIOS.***

Consiel
N'exécutez pas de programmes utilisant **des extensions d'interruption spécifiques à Intermec** ***sur votre PC. Si vous tentez de les exécuter sans le Simulator, ces programmes risquent de verrouiller votre PC et d'altérer le BIOS de votre système.***

INT 14H, Function 17H

Receive Next Buffer

Purpose: This function requests the protocol handler to receive the next data buffer from the communications port. Use this function to receive subsequent transmissions after INT 14H, function 60H, has established the link to the protocol handle.

Call With: AH = 17H Receive next buffer

DX = Port ID:
- 00H COM1
- 01H COM2, scanner port (2010 and 2050)
- 03H COM4

Return Value: AX = *nnnn* One of the standard status codes defined in Appendix A, "Status Codes."

See Also: PHTERM.CPP in Appendix B, "Sample Interrupt Programs."

INT 14H, Function 50H

Transmit Buffer

Purpose: This function transmits the contents of a data buffer using the selected communications port. You must install a protocol handler to use this interrupt.

Call With:

AH = 50H Transmit buffer

DX = Port ID:
- 00H COM1
- 01H COM2, scanner port (2010 and 2050)
- 03H COM4

ES:BX = Pointer to the data buffer structure defined in IM20LIB.H.

Return Value: AX = *nnnn* One of the standard status codes defined in Appendix A, "Status Codes."

See Also: PHTERM.CPP in Appendix B, "Sample Interrupt Programs."

INT 14H, Function 60H

Receive Buffer

Purpose: This function first establishes a link with the protocol handler and then receives the first record through the selected communications port.

Call With: AH = 60H Receive buffer

DX = Port ID:
- 00H COM1
- 01H COM2, scanner port (2010 and 2050)
- 03H COM4

ES:BX = Pointer to the IM_COM_DATA_BUFFER data structure defined in IM20LIB.H. The buffer must be larger than 256 bytes.

Return Value: AX = *nnnn* One of the standard status codes defined in Appendix A, "Status Codes."

Notes: You must install a protocol handler to use this interrupt.

Use this function only once at the start of your program.

See Also: PHTERM.CPP in Appendix B, "Sample Interrupt Programs."

INT 14H, Function 71H

Protocol Extended Status

Purpose: This function returns the protocol extended status for a selected communications port.

Call With:

AH = 71H Protocol extended status

DX = Port ID:
- 00H COM1
- 01H COM2, scanner port (2010 and 2050)
- 03H COM4

CX = Status buffer length

ES:BX = Pointer to the status buffer. See the file IM20LIB.H for details.

Return Value: AX = *nnnn* One of the standard status codes defined in Appendix A, "Status Codes."

See Also: EXSTAT.CPP in Appendix B, "Sample Interrupt Programs."

INT 14H, Function 84H

Cancel RX Buffer

Purpose: This function terminates the protocol handler's connection to the receiving buffer.

Call With: AH = 84H Cancel RX buffer

DX = Port ID:
- 00H COM1
- 01H COM2, scanner port (2010 and 2050)
- 03H COM4

Return Value: AX = *nnnn* One of the standard status codes defined in Appendix A, "Status Codes."

Notes: You must install a protocol handler to use this interrupt.

See Also: PHTERM.CPP in Appendix B, "Sample Interrupt Programs."

INT 14H, Function 85H

Cancel TX Buffer

Purpose: This function terminates the protocol handler's connection to the transmit buffer for a selected communications port.

Call With: AH = 85H Cancel TX buffer

DX = Port ID:
- 00H COM1
- 01H COM2, scanner port (2010 and 2050)
- 03H COM4

Return Value: AX = *nnnn* One of the standard status codes defined in Appendix A, "Status Codes."

Notes: You must install a protocol handler to use this interrupt.

See Also: PHTERM.CPP in Appendix B, "Sample Interrupt Programs."

INT 16H, Function B4H

Enable/Disable Expanded Keyboard Buffer

Purpose: This function enables or disables the expanded keyboard buffer. If it is enabled, the keyboard buffer is 269 bytes long. If it is disabled, the keyboard buffer is 16 bytes long.

Call With:

AH =	B4H	Enable / disable expanded keyboard buffer
AL =	00H	Get status only
	01H	Enable expanded keyboard
	02H	Disable expanded keyboard

Return Value:

AH =	00H	Always
AL =	00H	Command completed
	01H	Expanded buffer is already installed or already enabled
	02H	Expanded buffer is disabled
	03H	Cannot disable because expanded buffer is not empty
	04H	Invalid command

Notes: You must use INT 16H to read keycodes or first call the "Flush expanded key buffer" service to empty the buffer.

See Also: KEYBUFF.CPP in Appendix B, "Sample Interrupt Programs."

INT 16H, Function B5H

Insert String Into Expanded Keyboard Buffer

Purpose: This function inserts a string of up to 255 characters into the keyboard buffer with a single call to the BIOS. You can encode a passed string in two ways:

- Simple byte-aligned ASCII only
- Entire character keycode

Use the entire character keycode method to implement a unique character tagging scheme or to pass special keys such as **Ctrl**, **F1**, and **Esc**. The calling routine must supply the scan code.

Call With:

AH =	B5H	Insert string into expanded keyboard buffer
AL =	00H	Use byte-aligned ASCII string format with scan code lookup
	01H	Use word-aligned keycode format
	02H	Use byte-aligned ASCII string format without lookup
CX =		Number of characters to put into buffer
ES:BX =		Far pointer to the string

Return Value:

AH =	00H	Always
AL =	00H	Successful
	01H	Expanded buffer has not been installed
	02H	Keyboard buffer is full and the entire string was rejected
	03H	Invalid mode value
DX =	*nn*	Number of unused keys (*nn*) that remain in the buffer after the passed string has been inserted, or the number available prior to the call if the buffer cannot hold the string.

Notes: Pass strings instead of individual characters to eliminate label fragmentation and speed up the software interface between DECODES and the keyboard stream.

See Also: KEYBUFF.CPP in Appendix B, “Sample Interrupt Programs.”

INT 16H, Function B6H

Flush Expanded Keyboard Buffer

Purpose: This function initializes the expanded keyboard buffer from an unknown state. The contents of both the native and expanded buffers (all pending keycodes) are cleared and the expanded buffer is ready for new input from the keypad or from string and character insertions. This function has no effect if the expanded buffer is not enabled and keys in the native buffer are preserved.

Call With: AH = B6H Flush expanded keyboard buffer

Return Value: AH = 00H Always

AL = 00H Buffer is cleared
01H Expanded buffer is not installed

See Also: KEYBUFF.CPP in Appendix B, "Sample Interrupt Programs."

INT 16H, Function B7H

Enable/Disable Ctrl-Alt-Del Warm Boot

Purpose: This function blocks the user's attempt to warm boot the reader from the keypad.

Call With:

AH = B7H Enable / disable **Ctrl-Alt-Del** warm boot

AL = 00H Disable warm boot
01H Enable warm boot
02H Get status only

Return Value:

AH = General status
00H Valid service call
01H Invalid function number

AL = Keypad warm boot status
00H Warm boot is prohibited
01H Warm boot is permitted

Notes: This feature locks the application program to protect transient data from being lost during warm boot.

See Also: JBOOT.CPP in Appendix B, "Sample Interrupt Programs."

INT 78H, Function 06H

Restore UART Configurations

Purpose: This function restores the UART registers for COM1. Use this function to restore the UART parameters after the reader has gone through a suspend / resume cycle.

Call With:

AH = 06H Restore UART configurations

AL = 00H Save the current UART parameters, and get current UART configurations
- BL = 01H COM1
- 02H COM2, scanner port (2010 and 2050)
- BH = 00H
- CX = 0000H
- DX = 0000H

AL = 04H Restore the UART configuration
- BL = 01H COM1
- 02H COM2, scanner port (2010 and 2050)
- BH = 00H
- CX = 0000H
- DX = 0000H

AL = 06H Store the FCR parameter (FIFO control register)
- BL = 01H COM1
- 02H COM2, scanner port (2010 and 2050)
- BH = FCR value
- CX = 0000H
- DX = 0000H

Notes: The parameter values for the communications ports (01H and 02H) are correct as shown. COM4 is not allowed.

You can find complete register descriptions in generic data sheets for the 16x550 UART. You can find the FIFO Control Register (FCR) descriptions in your JANUS user's manual.

Notes: When power resumes after a suspend/resume cycle, reader services are restored in a manner transparent to client software. If your application uses the COM1 or COM2 UART directly for serial communications without the Intermec protocol handler software, your application must follow these guidelines:

16x550 UART The 16x550 UART prevents transparent restoration of COM1 or COM2 configurations when the reader is cycled through suspend/resume. Specifically, the 16x550's FCR is a write-only register that prohibits a transparent save operation prior to removing power from the UART. To prevent FCR-related problems in your communications application, use subfunction 06H as described below.

PC-Native UART You can ignore the UART restore issue altogether by using the UART in its PC-native mode (for example, without the FIFO buffering feature that is equivalent to a 16x450 or the original 8250 UART), since all other registers are readable and the system software manages them transparently.

Using Subfunctions If your application uses the FIFO feature of the 16x550, you can use subfunction 06H to store the FCR parameter (assuming all other UART registers are configured by the application). Once the FCR value is stored (a one-time operation), the entire set of UART registers are restored transparent to the application. Serial communications can survive all suspend/resume cycles for the duration of the application's run session.

Baud rates and their corresponding index values are as follows:

Baud Rate	Index Value	Baud Rate	Index Value
110	0	9600	6
300	1	19200	7
600	2	38400	8
1200	3	57600	9
2400	4	115000	10
4800	5		

Notes: The restore option is applied to the entire set of UART registers:

- FIFO Control Register FCR
- Divisor Latch least significant byte DLL, most significant byte DLL, and DLM baud rate selectors
- Interrupt Enable Register IER
- Line Control Register LCR
- Modem Control Register MCR
- Scratch Register SCR

The Receive Buffer Register (RBR) holding the last received data value (prior to entering the Suspend state) is not restored. The serial communications application currently controlling the COM port must handle this protocol recovery issue. Serial activity occurring at the time of a Suspend keeps the unit awake until a timeout (see Chapter 4, "Advanced Programming"). However, critical Suspend conditions can cause a Suspend before all serial activity is completed.

Client Software Requirements The power management (IPM) software normally calls the restore service. Thus, the COM1 or COM2 communications software does not need subfunction 04H. For testing purposes, any software client that wants to restore all preset (memorized) UART configurations can call subfunction 04H.

Before invoking the restore service, client software must protect the system by ensuring that either all COM interrupts are disabled (IRQ3 and IRQ4), or that the COM(n) interrupt service is initialized to begin processing serial data before the UART restore service returns to the calling software.

Return Value: When AL = 00H, the function returns:

AL =	00H	Successful
	01H	Invalid AX register
	05H	Invalid COM port number in BL register
	FFH	Undefined baud rate

AH = Baud rate table index value (see Notes)

Return Value:

BL = Interrupt enable register

BH = FIFO control register (returns the value set by subfunction 06 only, since you cannot read the FCR)

CL = DLL value (baud divisor LSB)

CH = DLM value (baud divisor MSB)

DL = LCR value (line control register)

DH = MCR value (modem control register)

When AL = 06H, the function returns:

AL = 00H Valid service
01H Invalid AX register
05H Invalid parameters (BL must be 01H, and DL must be 00H)

AH = 00H

BX = Unchanged

CX = Unchanged

DX = Unchanged

See Also: FIFO.CPP in Appendix B, "Sample Interrupt Programs."

INT 79H, Function 00H

LCD Display Contrast

Purpose: This function controls the LCD contrast. It does not interfere with the keypad user control.

Call With:

AH = 00H LCD display contrast

AL = 00H Increase or decrease contrast by one level
BL = 00H Increase
BL = 01H Decrease

AL = 01H Set contrast to an absolute value
BL = 00H to
1FH Indicates the absolute contrast value (0 to 31)
02H Restore contrast level
03H Return current contrast level

Return Value: AL = Absolute contrast value achieved. If AL = FFH, then the interrupt routine received bad parameters.

See Also: CONTRAST.CPP in Appendix B, "Sample Interrupt Programs."

INT 79H, Function 01H

LCD Display Backlight

Purpose: This function controls the LCD backlight. It does not interfere with the keypad user control.

Call With: AH = 01H LCD display backlight

AL = 00H Backlight OFF
01H Backlight ON
02H Backlight toggle
03H Return status
04H Enable or disable the auto-off feature
BL = 00H Disable
BL = 01H Enable
05H Set backlight timeout period
BL = 01H to 3CH 1 to 60 seconds
06H Return backlight timeout period
07H Return auto-off status

Return Value: When AL = 00H, 001H, or 05H, the return values are:

AL = 00H Good parameters were received
FFH Bad parameters were received

When AL = 02H, the return values are:

AL = 00H Backlight is OFF after toggle
01H Backlight is ON after toggle
FFH Bad parameters were received

When AL = 03H, 04H, or 07H, the return values are:

AL = 00H Disable
01H Enable
FFH Bad parameters were received

Return Value: When AL = 06H, the return values are:

AL =	0 to 60	Backlight timeout in seconds
	FFH	Bad parameters were received

See Also: BACKLITE.CPP in Appendix B, "Sample Interrupt Programs."

INT 79H, Function 02H

LCD Display Size Mode Manager

Purpose: This function establishes the display setup when in text mode.

Call With:

AH = 02H LCD display size mode manager

AL = 00H Initialize 80x25 or 40x25 text mode
- BH = 00H Video mode 0 (40 x 25)
- BH = 01H Video mode 1 (40 x 25)
- BH = 02H Video mode 2 (80 x 25)
- BH = 03H Video mode 3 (80 x 25)
- BL = 00H Scroll at line 25
- BL = 01H Scroll at line 16 (2010 and 2020)
- BL = 02H Scroll at line 8 (2010 and 2020)
- BL = 03H Scroll at line 13 (2050 and double height)
- CH = 00H Standard character height
- CH = 01H Double character height

AL = 01H Initialize 20x16 text mode (2010 and 2020)

02H Initialize 20x8 text mode (2010 and 2020)

03H Return current video mode, size mode, character height, and scroll mode

04H Initialize 10x16 text mode (2010 and 2020)

05H Initialize 10x8 text mode (2010 and 2020)

06H Enable or disable character height and scroll mode keys
- BL = 00H Disable
- 01H Enable

AL = 07H Returns whether the character height and scroll mode keys are enabled or disabled

Return Value: When AL = 03H, the return values are:

AH =	00H	Video mode 0
	01H	Video mode 1 (40x25)
	02H	Video mode 2 (80x25)
	03H	Video mode 3 (80x25)
AL =	00H	80x25 or 40x25 size mode
	01H	20x16 size mode (2010 and 2020)
	02H	20x8 size mode (2010 and 2020)
	03H	10x16 size mode (2010 and 2020)
	04H	10x8 size mode (2010 and 2020)
BH =	00H	Standard character height
	01H	Double character height
BL =	00H	Scroll mode 0
	01H	Scroll mode 1
	02H	Scroll mode 2

When AL = 07H, the return values are:

AL =	00H	Disabled
	01H	Enabled
	FFH	Bad parameters were received

When AL = any other value (except 03H and 07H), the return values are:

AL =	00H	Good parameters were received
	FFH	Bad parameters were received

See Also: PHTERM.CPP in Appendix B, “Sample Interrupt Programs.”

INT 79H, Function 03H

Viewport Move

Purpose: This function provides program and keypad control of the viewport. This service is available only in modes where the display buffer cannot be completely viewed within a 20x16 window.

This function does not apply to the JANUS 2050.

Call With:

AH = 03H Viewport move

AL = 00H Disable or enable movement keys
- BL = 00H Disable
- BL = 01H Enable

AL = 01H Absolute viewport move
- BX = Row number
- CX = Column number

AL = 02H Relative viewport move
- BH = Direction
  - 00H Left
  - 01H Right
  - 02H Up
  - 03H Down
- BL = Calling source
  - 00H Keypad
  - 01H Application
- CX = Distance
  - 00H Default distance

AL = 03H Set the viewport default step size
- BL = Video mode
- BH = Row default step size
- CX = Column default step size

AL = 04H Get the viewport default step size

AL = 05H Get the video, size, and position mode

Call With: AL = 06H Return whether viewport keys are enabled or disabled

Return Value: When AL = 01H, the return values are:

AX =	*nn*	Row number (*nn*) actually achieved
	FFH	Bad parameters were received
BX =	*nn*	Column number (*nn*) actually achieved
	FFH	Bad parameters were received

When AL = 02H, the return values are:

AX =	*nn*	Row number (*nn*) actually achieved
	FFH	Bad parameters were received
BX =	*nn*	Column number (*nn*) actually achieved
	FFH	Bad parameters were received

When AL = 03H, the return values are:

AL =	00H	Good parameters were received
	FFH	Bad parameters were received

When AL = 04H, the return values are:

AL =	*nn*	Row step size (*nn*)
	FFH	Bad parameters were received
BH =	*nn*	Row default step size (*nn*)
BX =	*nn*	Column step size (*nn*)
CX =	*nn*	Column default step size (*nn*)

Return Value: When AL = 05H, the return values are:

AH =	*nn*	Video mode (*nn*)
AL =	*nn*	Size mode (*nn*)
	FFH	Bad parameters were received
BX =	*nn*	Row number (*nn*)
CX =	*nn*	Column number (*nn*)

When AL = 06H, the return values are:

AL =	00H	Disabled
	01H	Enabled
	FFH	Bad parameters were received

See Also: PHTERM.CPP in Appendix B, “Sample Interrupt Programs.”

INT 7CH, Function 64H

Generate Sound

Purpose: This function generates a sound of specified pitch and duration.

Call With: AX = 64H Generate sound

CX = 00H

DX = 00H

ES:BX = Pointer to the IM_BEEP_PARAMS structure

The definition for this structure is as follows:

Byte	Description
0	Requested volume 0 = Off 1 = Quiet 2 = Normal 3 = Loud 4 = Very loud 5 = Default volume
1 to 2	Pitch (20 to 20,000 Hz)
3 to 4	Duration (1 to 65,000 ms)
5 to 6	Off time (1 to 65,000 ms)

Return Value: AX = *nnnn* One of the standard status codes defined in Appendix A, "Status Codes."

See Also: JBEEP.CPP in Appendix B, "Sample Interrupt Programs."

INT 7DH, Function 08H

Reader Wedge Control

Purpose: This function provides Reader Wedge control.

Call With:

AH = 08H Reader Wedge control

AL =
- 00H Clear abort callback address
- 01H Modify reader configuration
  - ES:BX Far pointer to a reader command string
  - CX Length of the reader string
- 03H Get input mode
- 06H Get label symbology
  - ES:BX Far pointer to a variable receiving the symbology
- 07H Get length of received string
  - CX
    - 0010H Label selected
    - 0020H Keyboard selected
    - 0001H COM1 selected
    - 0002H COM2, scanner selected (2010 and 2050)
    - 0008H COM4 selected
- 08H Get input status
- 0AH Get input from keyboard or scanned label (see IRL command A)
  - ES:BX Far pointer to IRL_PARAMS_S
- 0BH Receive input from keyboard (see IRL command K)
  - ES:BX Far pointer to IRL_PARAMS_S
- 0CH Receive numeric input from keyboard or scanned label (see IRL command N)
  - ES:BX Far pointer to IRL_PARAMS_S

Call With: AL = 0DH Receive input from any source in any format
(see IRL command V)
ES:BX Far pointer to IRL_PARAMS_S
CL 00H Edit disable
01H Edit enable
CH Beep mode:
00H Application beep
01H Wedge beep
02H No beep
DL Display flag:
00H Flag disable
01H Flag enable
DS:SI Far pointer to allowable input sources:
0000H No input source
0010H Label selected
0020H Keyboard selected
0001H COM1 selected
0002H COM2, scanner selected (2010 and 2050)
0008H COM4 selected
007FH All selected

0EH Receive input from a selected communications port
(see IRL command Y)
ES:BX Far pointer to IRL_PARAMS_S
CL 00H COM1 selected
01H COM2, scanner selected (2010 and 2050)
03H COM4 selected
DL Protocol mode:
00H No change
01H Protocol on
02H Protocol off
03H New end-of-message character
DS:SI Far pointer to the end-of-message character

10H Link to a selected communications port
ES:BX Far pointer to memory area used by Reader Wedge
CX 0000H COM1 selected
0001H COM2, scanner selected (2010 and 2050)
0003H COM4 selected
DX Number of bytes to allocate for the buffer array

Call With: AL = 1BH Receive input from selected source and place into buffer

- CX Allowed input source:
  - 0010H Label selected
  - 0020H Keyboard selected
  - 0001H COM1 selected
  - 0002H COM2, scanner port (scanner) selected (2010 and 2050)
  - 0008H COM4 selected
- DX Timeout value
- ES:BX Far pointer to the returned source:
  - 0000H No selection made
  - 0010H Label selected
  - 0020H Keyboard selected
  - 0001H COM1 selected
  - 0002H COM2, scanner port (scanner) selected (2010 and 2050)
  - 0008H COM4 selected
- DS:SI Far pointer to the input string

1EH Set serial protocol mode

- CL
  - 00H COM1
  - 01H COM2, scanner port (2010 and 2050)
  - 03H COM4
- CH Protocol mode:
  - 00H No change
  - 01H Protocol on
  - 02H Protocol off
  - 03H New end-of-message character
- DL End-of-message character

1FH Set abort callback address

- CX Input source:
  - 0010H Label selected
  - 0020H Keyboard selected
  - 0001H COM1 selected
  - 0002H COM2, scanner selected (2010 and 2050)
  - 0008H COM4 selected
  - 007FH All selected
- ES:BX Far pointer to the callback routine

Call With:

AL = 20H Set input mode
CX 0000H Programmer mode
0001H Wedge mode
0002H Desktop mode

22H Set standby wait mode
CX Timeout value

23H Unlink a communications port
CX = 0000H COM1
0001H COM2, scanner port (2010 and 2050)
0003H COM4

Return Value: When AL is one of the following values, the return value in register AX is one of the standard status codes defined in Appendix A, "Status Codes."

AL =				
	00H	0BH	10H	22H
	01H	0CH	1BH	23H
	06H	0DH	1EH	
	0AH	0EH	1FH	

When AL = 03H, the return values are:

AX = 00H Programmer mode
01H Wedge mode
02H Desktop mode

When AL = 07H, the return value is:

AX = The length of the string.

When AL = 08H, the return values are:

AX = 00H No selection made
10H Label selected
20H Keyboard selected
01H COM1 selected
02H COM2, scanner port selected (2010 and 2050)
08H COM4 selected
7FH All selected

Return Value: When AL = 20H, there is no return value.

Notes: Because the number of parameters passed to the IRL functions exceed the registers available, you must pass a structure variable of type struct IRL_PARAMS_S as shown in the following C code:

```
typedef struct
       {
       IM_USHORT timeout;
       IM_LENGTH_SPEC test_table[];
       IM_UCHAR mask_string[];
       IM_UCHAR *string;
       IM_USHORT *cmd_count;
       IM_DECTYPE *symbology;
       } IRL_PARAMS_S;
```

INT 7DH, Function 0AH

Get 2D Decoder Firmware Version

Purpose: This function returns all pertinent information about the revision level of the 2D Decoder and its firmware.

Call With: AH = 0AH Get 2D Decoder version

ES:BX = Far pointer to the client's Decoder response buffer (response buffer must be large enough to hold the response string)

Return Value: The return value in register AX is one of the standard status codes defined in Appendix A, "Status Codes."

Notes: The JANUS 2D upon boot queries the decoder for the firmware revision level. This function returns the string "UNKNOWN" until the decoder replies with the decoder revision level.

See Also: PL220VER.CPP in Appendix B, "Sample Interrupt Programs."

INT 7DH, Function 0CH

Open COM2

Purpose: This function opens COM2 for use by the 2D Decoder.

Call With: AH = 0CH Open COM2

Return Value: The return value in register AX is one of the standard status codes defined in Appendix A, "Status Codes."

Notes: The JANUS 2D upon boot opens COM2 for use by the 2D Decoder. When COM2 is being shared with other RS-232C devices (i.e., SAVI Interrogator), it is necessary to close COM2 prior to the other device using the port and open it backup prior to the 2D Decoder using COM2.

See Also: OPEN.CPP in Appendix B, "Sample Interrupt Programs."

INT 7DH, Function 0DH

Close COM2

Purpose: This function closes COM2 so other devices (i.e. SAVI Interrogator) can use COM2.

Call With: AH = 0DH Close COM2

Return Value: The return value in register AX is one of the standard status codes defined in Appendix A, “Status Codes.”

Notes: The JANUS 2D upon boot opens COM2 for use by the 2D Decoder. When COM2 is being shared with other RS-232C devices (i.e., SAVI Interrogator), it is necessary to close COM2 prior to the other device using the port and open it backup prior to the 2D Decoder using COM2.

See Also: CLOSE.CPP in Appendix B, “Sample Interrupt Programs.”

INT 7EH, Function 01H

Get Application Break Status

Purpose: This function determines if the application break sequence was entered on the keypad.

Call With:

AH = 01H Get application break status

ES:BX = Far pointer to the client's response buffer structure (two bytes, R0 and R1)

Return Value:

AH = General status:
- 00H Valid service call
- 01H Invalid function number

R0 = Keyboard scanner CPU (KSCPU) communication status
- 00H Successful
- 01H Timeout occurred during the execution of the transaction
- 02H No KSCPU response (timeout on first attempted transmit)
- 03H KSCPU is busy
- 04H Forced the transaction to abort
- 05H KSCPU failed to process an acknowledged command within the timeout period

R1 = Application break status
- 00H Application break request has not been made
- FFH Application break request has been made

Notes: Applications that support the application break feature must either poll the unlock status using this service or implement a **Ctrl-C** handler (INT 23H) that queries using this service. In the latter case, the operator is required to enter **Ctrl-C** on the keypad after entering the application break sequence and resume the unit by pressing the **1/0** key.

To enter an application break sequence

1. Press (I/O) to turn off the reader.
2. Press (F9) - [2] - ◆ at the same time.
3. Press [1]. This sets the reader's application break bit.
4. Press (I/O) to turn on the reader.

This sequence resets the application break request bit if the bit is TRUE. A new application that monitors this bit can execute without terminating on the same condition that terminated the previous application.

See Also: APPBREAK.CPP in Appendix B, "Sample Interrupt Programs."

INT 7EH, Function 03H

Get Keypad Scanner Version

Purpose: This function returns all pertinent information about the revision level of the KSCPU and its firmware.

Call With:

AH = 03H Get keypad scanner version

ES:BX = Far pointer to the client's KSCPU response buffer (five bytes, R0 to R4)

Return Value:

AH = General status
- 00H Valid service call
- 01H Invalid function

R0 = KSCPU communication status
- 00H Successful
- 01H Timeout occurred during the execution of the transaction
- 02H No KSCPU response (timeout on first attempted transmit)
- 03H KSCPU is busy
- 04H Forced transaction abort
- 05H KSCPU failed to process an acknowledged command within the timeout period

R1 = KSCPU minor revision level

R2 = KSCPU major revision level

R3, R4 = Workspace padding (required)

See Also: KEYSTAT.CPP in Appendix B, "Sample Interrupt Programs."

INT 7EH, Function 04H

Get Keypad Map Table ID and Status

Purpose: This function returns the keypad map table ID and status.

Call With: AH = 04H Get keypad map table ID and status

ES:BX = Far pointer to the client's KSCPU response buffer (seven bytes, R0 to R6)

Return Value: AH = General status
- 00H Valid service call
- 01H Invalid function number

R0 = KSCPU communication status
- 00H Success transaction
- 01H Timeout occurred during the execution of the transaction
- 02H No KSCPU response (timeout on first attempted transmit)
- 03H KSCPU is busy
- 04H Forced transaction abort
- 05H KSCPU failed to process an acknowledged command within the timeout period

R1 = Keypad ID number

R2 = Table validation pass/fail
- FFH Pass
- 00H Fail

R3, R4 = Keypad checksum/CRC 16-bit word, R3 is LSB

R5, R6 = Workspace padding (required)

See Also: KEYSTAT.CPP in Appendix B, "Sample Interrupt Programs."

INT 7EH, Function 05H

Enable/Disable KSCPU Keyclick

Purpose: This function turns the keypad clicker ON or OFF.

Call With:

AH = 05H Enable / disable KSCPU keyclick

AL = 00H Turn the keyclick OFF
01H Turn the keyclick ON
02H Get status only

ES:BX = Far pointer to the client's KSCPU response buffer (four bytes, R0 to R3)

Return Value:

AH = General status
00H Valid service call
01H Invalid function number

R0 = KSCPU communication status
00H Successful
01H Timeout occurred during the execution of the transaction
02H No KSCPU response (timeout on first attempted transmit)
03H KSCPU is busy
04H Forced transaction abort
05H KSCPU failed to process an acknowledged command within the timeout period

R1 = Keyclick status
00H Keyclick disabled
FFH Keyclick enabled

R2, R3 = Workspace padding (required)

See Also: KEYCLICK.CPP in Appendix B, "Sample Interrupt Programs."

INT 7EH, Function 06H

Enable/Disable Numeric Keypad

Purpose: This function toggles the numeric keypad modes of the reader's keypad. When disabled, normal QWERTY scan codes are sent by the numeric keys. When enabled, the PC KB-102 numeric keypad scan codes are sent instead.

Call With:

AH = 06H Enable / Disable numeric keypad

AL =
- 00H Disable
- 01H Enable with NUM LOCK on
- 02H Enable with NUM LOCK off
- 03H Get status only

CL =
- 00H Prohibit mode-switching from the keypad
- 01H Permit mode switching from the keypad
- 02H Reserved
- 03H Get status only

ES:BX = Far pointer to the client's KSCPU response buffer (five bytes, R0 to R4)

Return Value:

AH = General status
- 00H Valid service call
- 01H Invalid function number

R0 = KSCPU communication status
- 00H Successful
- 01H Timeout occurred during the execution of the transaction
- 02H No KSCPU response (timeout on first attempted transmit)
- 03H KSCPU is busy
- 04H Forced transaction abort
- 05H KSCPU failed to process an acknowledged command within the timeout period

Return Value: R1 = Number pad status
- 00H Disabled
- FFH Enabled

R2 = Permit status
- 00H Switching the number pad is prohibited
- FFH Switching the number pad is permitted

R3, R4 = Workspace padding required

Notes: The R2 permit status only affects the manual keypad method of switching the number pad ON and OFF. Even if the numeric keypad is not manually permitted, you can use program-controlled mode changes (regardless of the previous state).

See Also: NUMPAD.CPP in Appendix B, "Sample Interrupt Programs."

INT 7EH, Function 08H

Enable/Disable Ctrl Key

Purpose: This function enables or disables the **Ctrl** key on the reader's keypad.

Call With:

AH = 08H Enable / disable **Ctrl** key

AL = 00H Disable
01H Enable
02H Get status only

ES:BX = Far pointer to the client's KSCPU response buffer (four bytes, R0 to R3)

Return Value:

AH = General status
00H Valid service call
01H Invalid function number

R0 = KSCPU communication status
00H Successful
01H Timeout occurred during the execution of a transaction
02H No KSCPU response (timeout on first attempted transmit)
03H KSCPU is busy
04H Forced transaction abort
05H KSCPU failed to process an acknowledged command within the timeout period

R1 = **Ctrl** key status
00H Disabled
FFH Enabled

R2, R3 = Workspace padding (required)

Notes: The system BIOS does not detect a **Ctrl** key press when you disable the **Ctrl** key with this interrupt.

See Also: CTLKEY.CPP in Appendix B, "Sample Interrupt Programs."

INT 7EH, Function 10H

Enable Viewport

Purpose: This function enables the viewport so that it automatically follows the cursor. This function does not apply to the JANUS 2050.

Call With: AH = 10H Enable viewport

Return Value: AX = *nnnn* One of the standard status codes defined in Appendix A, "Status Codes."

See Also: VP_CUR.CPP in Appendix B, "Sample Interrupt Programs."

INT 7EH, Function 11H

Disable Viewport

Purpose: This function disables the viewport from automatically following the cursor. This function does not apply to the JANUS 2050.

Call With: AH = 11H Disable viewport

Return Value: AX = *nnnn* One of the standard status codes defined in Appendix A, "Status Codes."

See Also: VP_CUR.CPP in Appendix B, "Sample Interrupt Programs."

4

Advanced Programming

This chapter explains the various reader input modes and how to use the power management software.

Reader Input Modes

Reader input modes affect how scanned input is interpreted and stored by the reader. You can control input modes by:

- Using the reader commands described in your JANUS user's manual.
- Using the im_get_input_mode and im_set_input_mode commands described in Chapter 2, "C Language Support."

There are three different input modes:

- Wedge mode
- Programmer mode
- Desktop mode

Wedge Mode

In Wedge mode, keypad and label inputs go into the keyboard buffer with minimal filtering. Wedge mode is the default mode at the DOS prompt after reader services are loaded (RSERVICE.EXE). Use this mode when the reader is running an application that uses the Virtual Wedge. For more information about the Virtual Wedge, see Chapter 1, "Getting Started."

When the reader is in Wedge mode, use standard input functions such as `getch` to retrieve keypad or label input. Do not use the methods described for other modes.

Wedge mode does not take full advantage of the reader power management capabilities. You will probably notice reduced battery life when the reader is in Wedge mode.

Programmer Mode

In Programmer mode, keypad input is echoed to the screen as the keys are pressed. Any reader commands that you enter are executed and saved. Use this mode when the reader is executing Interactive Reader Language (IRL) commands such as im_irl_a or im_irl_k.

Desktop Mode

In Desktop mode, the application is responsible for retrieving and displaying keypad input. Use this mode when you need detailed information about the pressed key.

Use the input function im_receive_input to capture the keys pressed. Each character returned uses the structure IM_KEYCODE, described in IM20LIB.H. This structure consists of four bytes: the ASCII code, the scan code, and two bytes of keyboard flags (Shift, Control, Alt).

Differences in Input Modes

Programmer mode and Desktop mode differ only in how they handle keypad inputs. In programmer mode, keypad entry terminates when you press the **Enter** key. In desktop mode, keypad entry terminates each time you press a key.

Do not use functions such as `getch` to retrieve input when the reader is in Desktop or Programmer modes. The internal software intercepts any input before functions such as `getch` can receive it, and the application waits forever on that instruction.

In Wedge mode, label input is placed directly into the keyboard buffer. In Programmer and Desktop modes, label input is not placed into the keyboard buffer. It is, however, retrieved in a manner similar to the way keyboard data is retrieved. An example of handling reader input is given below.

Note: *For the reader to function as a bar code reader, LOADSYMB.EXE must be loaded. RDG.BAT loads both reader services (RSERVICE.EXE) and the decode algorithms (LOADSYMB.EXE).*

Example: Input Modes

```
// This example demonstrates reader input from either the keyboard or a label. Additional
// sources (or a single source) can be given. All possible sources, including COM ports,
// can be selected using IM_ALL_SELECT.
//
// In programmer mode, the im_receive_input() function returns after the Enter key is
// pressed. In desktop mode, the im_receive_input() function returns after each
// key is pressed.
//
// This method is used primarily to receive input from multiple sources when you do not
// know the input source.

char    input[80];                    // Input buffer (user sets the length)
IM_ORIGIN      source;                // Source(s) where input is to come from
IM_DECTYPE     symbol;                // Symbology (for label input)
IM_STATUS      istatus
im_set_input_mode(IM_PROGRAMMER);     // Can also be IM_DESKTOP (but not IM_WEDGE)

istatus = im_receive_input
               (im_keyboard_select | im_label_select,
               im_infinite_timeout,
               &source, input);
if (source  = =  IM_KEYBOARD_SELECT) {
       printf("Input from keyboard\n");
}
if (source  = =  IM_LABEL_SELECT) {
       printf("Input from scanner\n");
}
printf("%s\n", input);
```

Receiving Serial Data

There are two methods for receiving serial data:

- Reader service routines that allow multiple inputs
- Communications routines that use the INT 14H BIOS extensions

The method you use depends on how you want to receive keypad and label data. Both of these methods are valid in Desktop mode and Programmer mode. With minor modifications, these methods will work in Wedge mode.

Caution
Do not mix reader service routines with communications routines: a usable response is not returned.

Conseil
Ne mélangez pas les routines de révision lecteur à des routines de communication: aucune réponse utilisable ne sera renvoyée.

Reader Services Routines

You can use reader service routines that include communications port inputs by following these application requirements:

- At the beginning of your application, link the reader service input functions using a call to im_link_comm. At the end of your application, unlink them with a call to im_unlink_comm.
- Enable or disable the protocol using im_serial_protocol_control. The default is to have the protocol enabled.
- Clear the receive buffer by disabling and then enabling the protocol. Do not use im_cancel_rx_buffer when using this method because it stops the reader from communicating.

Example: Reader Services

```
// This example shows how to accept input from either the keyboard or from a serial
// port. The function im_receive_input( ) is given two sources: keyboard or COM1.
// Additional sources (or a single source) can be given. Use IM_ALL_SELECT to select
// all possible sources (including label input).
//
// This program waits for input, and then makes it available by calling
// im_receive_input( ).
//
// Use this method when you want to receive input from multiple sources and you don't know
// the input source.
//
// This is not a complete program, but it demonstrates the required function calls.

char    input[300];    // Input buffer (user sets the length)
char    com_buff[300]; // Used by Communications link only
IM_ORIGIN      source; // Source(s) where input is to come from
IM_STATUS      status; // Results of call

im_set_input_mode(IM_PROGRAMMER);     // In this example, the mode might be IM_DESKTOP
                                      // (but not IM_WEDGE).  If the reader is in
                                      // IM_WEDGE mode, use a function like gets( )
                                      // to retrieve keyboard and label input.
                                      // The only sources that can be ORed together in
//                                    // IM_WEDGE mode are COM ports.

im_link_comm(IM_COM1, com_buff, 300); // Link to reader services input routines

// Wait for input from either the keyboard or from COM 1
status = im_receive_input
               (IM_KEYBOARD_SELECT | IM_COM1_SELECT,
               IM_INFINITE_TIMEOUT,
               &source,
               input);

// Data is now available in the buffer input



if (status == IM_RW_SUCCESS)
       im_transmit_buffer(IM_COM1, strlen(input), input, 1000);    // Transmit data with
                                                                   // a 1-second timeout
im_unlink_comm(IM_COM1);              // Unlink from reader services input routines
```

Communications Routines

You can use communications routines with the INT 14H BIOS extensions by following these application requirements:

- Do not link your application with the reader service input functions (do not call im_link_comm).
- Use either im_receive_buffer or im_receive_buffer_noprot to select communications with or without protocol.
- Use im_cancel_rx_buffer to clear the receive buffer.

Example: Communications Routines

```
// This example shows how input is requested from a single source.  This function waits
// for input and then puts it into the specified buffer.
// In programmer mode, keyboard entry terminates when you press the Enter key. In desktop
// mode, keyboard entry terminates when you press each key.

IM_UCHAR       input[300];    // Input buffer (user sets the length)
IM_USHORT far  num_chars;
IM_DECTYPE     symbol;

im_set_input_mode(IM_PROGRAMMER);     // Could be any mode

im_receive_buffer(IM_COM1, 300, input, 1000, &num_chars);   // Get serial input (timeout)
                                                            // is 1-second)
im_transmit_buffer(IM_COM1, strlen(input), input, 1000);    // Transmit data with a
                                                            // 1-second timeout
```

Transmitting Serial Data

With Intermec protocol handlers, you transmit serial data by calling the communications routines included with the PSK. These routines are im_transmit_buffer, im_transmit_buffer_no_wait, and im_transmit_buffer_noprot. These communications routines all use the INT 14H BIOS extensions.

You can transmit a maximum of 254 bytes in one record. Make the buffer size somewhat larger (300 bytes is typically used) to allow adequate memory for protocol characters that might also be transmitted. If the buffer is too small, you will overwrite other memory locations.

Communications Port UART Modes

If you turn off your JANUS reader while running a communications application that uses COM1 or COM2, the reader cannot save the UART FIFO control register. The reader uses the UART restore configuration setting to reset the FIFO control register when the reader is turned back on. You need to set a restore value if the default value, 0 hex, is not appropriate for your reader. You must set a restore value in these two cases:

- Your application is using COM1 without using an Intermec protocol handler.
- Your application is using COM2 and changes the reader's COM2 UART FIFO control register to use UART 16x550 mode. See the communications application's manual to determine if it uses UART 16x550 mode.

You can also reset the FIFO control register using software interrupts. For more information, see INT 78H, function 06H, in Chapter 3, "Software Interrupts."

Display "Color"

You can set the reader display to various foreground and background shades. A SMARTMAP™ scheme compares and adjusts the foreground and background gray values on a character-by-character basis to produce adequate display contrast on monochrome LCD panels. With this scheme, commercially written color software packages can be used on a monochrome display with acceptable differentiation of colors using shades of gray.

The default color setting on a PC is white on black. However, the best default setting on a monochrome display is black on white. For this reason, setting a color to black actually sets white on black, and setting a color to white actually sets black on white.

To display colors other than just black and white, you should experiment with various color combinations to find an appropriate setting. The shade of a specific foreground color varies depending on the background color setting.

Borland C / C++ changes the foreground color by using textcolor(int *newcolor*) where *newcolor* is an integer between 0 and 15. Adding 128 to *newcolor* sets the most significant byte and makes the character blink. The background color is set using textbackground(int *newcolor*) where *newcolor* is an integer between 0 and 7. These colors are only applicable when using functions that do direct video output (such as cprintf).

Microsoft C / C++ changes the foreground color by using _settextcolor(short *index*) where *index* is an integer between 0 and 15. Adding 16 to *index* sets bit 4 and makes the character blink. The background color is set using _setbkcolor(long *index*) where *index* is an integer between 0 and 7. These colors are only applicable when using functions that do direct video output (such as _outtext).

Advanced Power Management

The reader power management subsystem provides Advanced Power Management (APM) compliant BIOS software functionality using the VLSI SCAMP II chip set.

The specific components of power management are:

- BIOS Power Management (PM) software with the APM software interface.
- Intermec Power Management (IPM) software.
- VLSI SCAMP chipset Power Management Unit (PMU) hardware (VL82C322).

APM is a layered, cooperative environment that allows applications, operating systems, and the system BIOS to work together toward the goal of reducing power consumption. The standardized flow of information and control across the layers facilitates interface requirements. The primary purpose of an APM implementation is to provide increased productivity and greater system availability by extending the life of system batteries without degrading performance.

Power Management Software Layers

Power management is provided by two software layers:

- System BIOS interface
- Application interface

The system BIOS interface is analogous to the power management methods implemented on current generation laptop systems, where automatic system functions temporarily shut down hardware components in order to prolong battery life.

The application interface improves power management by letting an application actively determine operational conditions and directly control hardware power consumption.

System BIOS Interface

The system BIOS is the lowest layer of power management software in the system. It is resident in the reader at all times. The system BIOS is capable of controlling power management functionality without any support from the operating system or application.

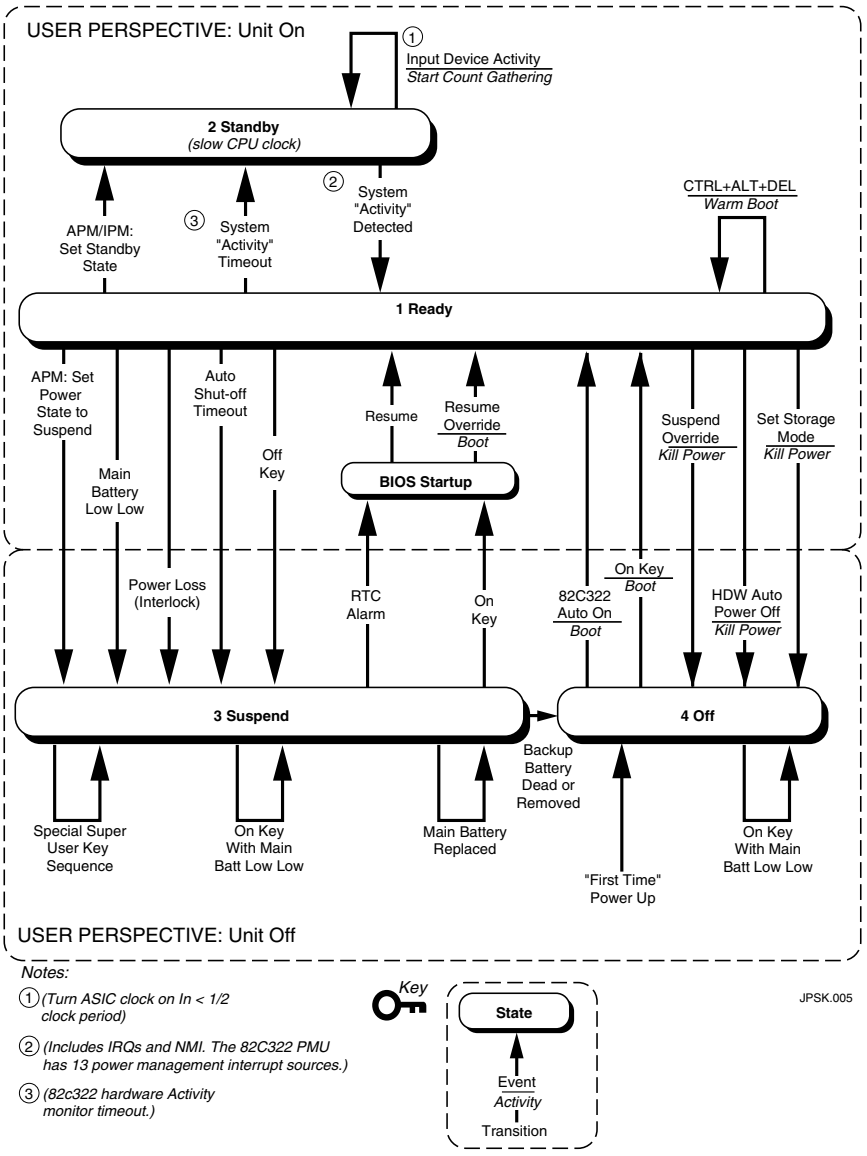
Application Interface

APM provides a cooperative interface between the system BIOS and other system or application software that makes the other software a partner in the power management process. Once this cooperative interface is established, the system BIOS informs its partner of significant power-related events and relies on its partner to actively participate in the power management process.

The application layer assists the power management function by providing information that only the application can easily ascertain. Applications are not required to support the power management function, but they can greatly increase its effectiveness. Under DOS in particular, the application is often in the best position to know when the reader is idle and awaiting input.

Some of the system BIOS functions can be called regardless of whether or not the cooperative power management interface has been established between the BIOS and the caller. However, some functions, particularly those related to notification of power events, do not operate until the cooperative interface is established. POWER.EXE provides a default cooperative connection to improve JANUS reader performance.

Figure 4-1 Power States



The APM Power States

APM supports four distinct power states: Ready, Standby, Suspend, and Off. Their relationships are depicted in Figure 4-1. Three of these states (Ready, Standby, and Off) apply both to individual system components and to the system as a whole. The Suspend state is a special low power condition that applies to the system as a whole, but not to the individual components.

Usually, your application only distinguishes between two apparent conditions: On and Off, where the indicator is the display. In the On condition, the display is active and the power management is in either the Ready or the Standby state. In the Off condition, the display is inactive, and power management is in either the Suspend or the Off state.

The system and devices change power states either by explicit command or automatically (based on APM parameters and system activity). The power capabilities of devices differ, and some devices may not be capable of achieving all states. Additionally, some devices may have unique, built-in power management that is invisible to the system, and therefore outside the scope of the APM specification.

Ready

In the Ready state, the system or device is fully powered up and available. The APM definition of Ready does not differentiate between active and idle conditions; it only indicates that the system or device is fully powered up. The reader enters the Ready state when a device raises a hardware interrupt or accesses any controlled device.

Standby

Standby is an intermediate system-dependent state that provides some power conservation. The reader enters the Standby state when the CPU is idle and there is no device activity (such as when an application is waiting for input). All data and operational parameters are preserved when the machine is in the Standby state. The CPU is powered but the clock runs very slowly (this does not affect the DOS time and date or any BIOS functions: it only conserves power).

Suspend

The Suspend state is the lowest level of power consumption available that preserves operational data and parameters. The reader enters the Suspend state by either the system BIOS or the software above the BIOS. The system BIOS might place the system into the Suspend state without notification if it detects a situation that requires an immediate response (such as the battery entering a critically low power state). When the system is in the Suspend state, computation is not performed until normal activity is resumed. Power is removed from the CPU. Resumption of activity does not occur until signaled by an external event such as pressing the **1/0** key.

Off

In the Off state, the unit is powered down and inactive. Data and operational parameters are not preserved (except for PC card drives that contain their own backup mechanisms and are preserved as long as they are intact). The reader enters the Off state when the operator selects storage mode (from the boot loader screen) or when all of the batteries are dead.

Power State Transitions

Figure 4-1 earlier in this chapter illustrates the methods and events used to transition between the indicated power states. States above the dotted line appear as On to the application, with the unit fully operational. States below the dotted line appear as Off to the application (the display is blank and a delay occurs when the unit is reactivated by pressing the **1/0** switch).

When the unit appears to be on, it might be in either the Ready or Standby state. In the Ready state, it is indeed on and fully powered. In the Standby state, the hardware is fully powered but the CPU clock is slowed, maintaining only memory refresh capability. To applications, the CPU clock appears to be stopped. The Standby state reduces battery current dramatically.

An exit from Standby occurs during any system activity that might require attention from the application. That is, not on BIOS clock ticks and not at the beginning of count gathering for bar code reading, although the end of the read generates an interrupt to direct the application to process the label.

When the unit appears to be Off, it is usually in the Suspend state. It is in the Off state only when both the main and backup batteries are dead or removed, or when placed into Storage mode from the bootstrap screen.

Figure 4-1 lists the events that control transitions between Ready and the Suspend or Off states. In normal operation, the unit transitions from Suspend to Ready when the **1/0** key is pressed and transitions back to Suspend on either an activity timeout or when the **1/0** key is again pressed.

While the reader is in the Suspend state, you can use the resume override key sequence to exit from Suspend and activate the boot screen. From the boot screen, you can select the following actions:

- Resume as long as the system state remains valid
- Reboot the reader, which transitions to On
- Select storage mode, which transitions to Off
- Use the other boot screen functions

Similar to an application, the CPU is in a Ready or Standby state while in the boot screen.

Power State Cooperation

APM provides an industry standard interface for cooperation among calling applications. The composite PM, BIOS, APM, and IPM provides two modes of operation:

- Transparent mode, where the BIOS, PMU, and PM cooperate to detect idle conditions and change power states by their set of rules.
- Nontransparent mode, where the currently operating application (client subsystem) cooperates in controlling the system power state and uses its knowledge of when it is waiting for activity. In this mode, the currently operating application can wait in a lower power state.

Basic power management capability is always present in the reader BIOS-PMU interaction. The reader reduces CPU power when it observes no system activity for a period of time, such as when the reader is sitting at the DOS prompt or waiting for operator input.

Basic power management uses the Standby and Auto-shutoff states. System activity is detected by monitoring for key press functions, display change activity, and all interrupts except the BIOS clock tick. In addition, the PMU hardware activity timeouts are set to change automatically to lower power states when there is no interrupt activity.

The Reader Wedge uses an IPM interface to control power management. For example, with IRL programs (which have no explicit PM interface), the Reader Wedge services know when the IRL program is waiting for input and that it can wait in a lower power mode. The Intermec configuration manager (IC.EXE) provides setup parameters for this level of power management.

Your application can explicitly control the power states by connecting to the APM interface with the interface connect function. This method provides the best use of the reader power since the application knows when it is idle and therefore can wait in a Standby or Suspend state depending upon response time requirements.

Applications that use the reader services interface have equally good power management without having to be explicitly concerned with power management functionality. The APM interface provides for polling by the application client to include power management information from the lower BIOS level software.

Finally, POWER.EXE (a Microsoft product) is an APM client that monitors system usage parameters to determine when an application is idle. POWER.EXE is a TSR program that provides better power management for off-the-shelf programs that have no APM knowledge. POWER.EXE also takes care of correcting the system date and time when the unit has been suspended overnight or longer. When the reader is shipped, POWER.EXE is installed and enabled.

The APM interface can only support one connected client at a time, so POWER.EXE must be disabled if an application wants to use the APM interface. You can disable POWER.EXE by placing the following command line in the AUTOEXEC.BAT file (or by executing the command at the DOS prompt, prior to loading an application):

```
POWER OFF
```

Programs that replace POWER.EXE must implement date correction algorithms when the reader is to be left suspended overnight or longer.

IPM Interface Functionality

The IPM interface is used by reader services and system functions to control power management state changes. It is transparent to application and operator controls. Both the IPM and APM interface cooperate to prevent the CPU from leaving the Ready state (except for critical Suspend transitions as discussed below).

The IPM interface provides Suspend and Resume functions for both normal and critical conditions:

- Normal Suspend
- Normal Resume
- Critical Suspend
- Critical Resume

Normal Suspend

The reader enters a normal Suspend transition when the front panel **1/0** key is pressed or the duration of the current activity exceeds the timeout parameter.

A normal Suspend only occurs when PM is enabled. Functions in process are completely restored following a normal Suspend.

Normal Resume

The reader enters a normal Resume transition when the front panel **1/0** key is pressed.

A normal Resume transition follows a normal Suspend transition. A normal Resume is not possible unless there has been a prior valid normal Suspend to save the system state. If the system state is not valid, a normal Resume forces a reboot. The operator can use the boot screen function to dump the system memory state (through the communications port) to a host prior to rebooting.

Critical Suspend

The reader enters a critical Suspend transition when the battery is removed, the main battery is low, or the front panel **1/0** key is pressed. The reader attempts to enter a normal Suspend transition before entering a critical Suspend transition.

Several warning periods occur prior to entering a low main battery state. Initially, the battery icon flashes as the battery begins to reach end of charge. Next the system emits warning beeps. Automatic shutoff with fully saved states occurs at a somewhat lower charge level. Certain system events (such as transmitting data using RF or a laser scanner) can add enough load to a low battery charge to cause an immediate transition to the low battery cutoff critical Suspend state.

A critical Suspend transition is always enabled. The application software must deal with the subsequent resume as effectively as possible. Functions in process might not be completely restored following a critical Suspend.

Critical Resume

The reader enters a critical Resume transition when the front panel **1/0** key is pressed or the battery is replaced.

A critical Resume transition follows a critical Suspend transition. A critical Resume is not possible unless there has been a prior valid critical Suspend to save the system state. If the system state is not valid, a critical Resume forces a reboot. The operator can use the boot screen function to dump the system memory state (through the communications port) to a host system prior to a reboot.

APM Interface Functionality

Programs that need to connect to and use the APM interface should observe the following suggested protocol:

- Turn off POWER.EXE (at the DOS command line) prior to connecting with the APM interface.
- Issue an APM installation check call. This call returns information on the current APM functionality (such as whether power management is currently enabled or disabled).
- Issue an APM interface connect call. If the return from this call shows an already connected client (such as POWER.EXE mentioned above), the application should terminate. The APM interface only allows one connected client.
- Issue an interface disconnect call, prior to exiting (for example, returning to the DOS prompt), so that another client can be connected.

Note: *For an example of APM interface protocol as discussed above, see the application listing FIFO.CPP in Appendix B, "Sample Interrupt Programs."*

You must enable power management for the interface connect call to be successful, otherwise the CPU stays in the Ready state until the operator presses an **1/0** key to transition to the Suspend state. Certain system activities (such as file access to the PC card) temporarily disable PM to protect their integrity. Your application might have its own reasons to disable PM. You can use the enable/disable power management call to enable PM.

When PM is enabled, your application can initiate a get PM event call, and either use the result to change states or ignore the PM event call, in which case the unit stays in the Ready state (unless reader services is active to prompt state changes). A get PM event returns a battery status value that your application can use.

When your application is connected as an APM client, it is notified during get PM event calls when a system Suspend condition occurs. APM requires your client application to complete current processes, issue a set power state command, and enter the Suspend state (effectively turning the unit off) as soon as possible.

Certain system events, such as low battery, cannot wait for this process to occur and instead cause an immediate transition to Suspend. The resume following each Suspend includes, as the first PM event response, the fact of resume and the resume type: normal or critical. After a critical resume occurs, your application must continue to operate as effectively as possible.

You can use power state commands to transition from Ready to Standby or Suspend. The APM interface coordinates with IPM to allow transitions from Ready only when both interfaces allow it. This coordination also applies to CPU Busy and CPU Idle. Either interface can keep the CPU in Ready, and both must allow other states.

A transition to Suspend stops the time and date of the system clock. Upon each resume, the IPM interface restores the time function from the real-time clock. Upon resume in some situations, such as when the unit is suspended over midnight or when the date changes, the APM client must restore the DOS date function by using the real-time clock and date. POWER.EXE restores the date when it is active.

Note: *For an example of how to enter the APM interface registers from the DOS command line and then connect to the APM interface, see the application listing APMIF.C in Appendix B, "Sample Interrupt Programs."*

Special APM BIOS Calls

APM requires a real mode interface and is implemented with an extension to INT 15. The system BIOS must operate in either real mode or virtual-86 mode on 80386 and later processors. Since the INT 15H processor instruction normally disables the interrupts, the system BIOS typically expects to be entered with interrupts disabled. However, the BIOS routines should not depend on any particular setting and should explicitly enable and disable interrupts as necessary. To avoid reentering INT 15H, an interrupt service routine (ISR) should not use any of the APM functions.

To better support the protected modes of 80286 and later processors, the system BIOS optionally supports a 16- and 32-bit protect mode interface directly callable from protected mode. First, you must initialize the protected mode interfaces using the real mode INT 15H AH = 53H. Systems not supporting APM return from this call with the carry flag (CY) set and the AH register equals 86H.

You must initialize 16- and 32-bit protected mode interfaces using the appropriate protected mode 16-bit interface connect or protected mode 32-bit interface connect functions.

If an error condition is detected when processing a system BIOS function, the function returns to the caller with the carry flag set and an error code in the AH register. The carry flag is cleared upon return from any successful call, and the content of the AH register is dependent on the particular call.

This interface adopts a convention for identifying a device class and specific subclasses within that class to use in direct device control. For example, a class might be all disk devices and the subclasses could be the physical unit numbers. APM calls take this parameter in a word-length register where the most significant byte (MSB) is the device class and the least significant byte (LSB) is the device subclass. This parameter is described in the individual interface descriptions.

The APM device class and subclass IDs are defined using BX = Device ID as follows:

Device Class ID		Device Subclass ID	
00XXH	System	00H	System BIOS
		01H	All devices power managed by the system BIOS
01XXH	Display	n/a	
02XXH	Secondary Storage	n/a	
03XXH	Parallel Ports	n/a	
04XXH	Serial Ports	n/a	
0500H to FFFFH	Reserved	n/a	

Where XXH indicates the physical device number (zero-based).

Where FFH indicates all devices in this class.

The following pages list the special APM system BIOS software interrupts that are unique to power management.

APM Installation Check

Purpose: This function determines if the system BIOS supports the APM functionality.

Call With: AH = 53H

AL= 00H Do APM installation check

BX = 0000H System BIOS

Return Value: If the function is successful:

Carry flag = Clear

AH = Major version number (in BCD format)

AL = Minor version number (in BCD format)

BH = ASCII P character

BL = ASCII M character

CX = Bit 0 = 1 if the BIOS supports 16-bit protected mode
Bit 1 = 1 if the BIOS supports 32-bit protected mode
Bit 2 = 1 if CPU Idle call slows processor clock speed
Bit 3 = 1 if BIOS Power Management is disabled
All other bits reserved

If the function is unsuccessful:

Carry flag = Set

AH = 86H APM is not present

Notes: The interpretation of bit 2 is documented under the CPU Busy call description.

This call is only available using the real mode INT 15H interface.

CPU Busy

Purpose: This function informs the system BIOS that the system is now busy and should continue at full speed. Some system implementations may only be able to slow the CPU clock rate and return in response to the CPU Idle call. The system BIOS usually restores the CPU clock rate to its normal rate when it recognizes increased system activity (typically interrupt driven), but it may be unable to do so when interrupts are hooked by external software and do not invoke BIOS routines.

You can ensure the system is running at full speed by using the CPU Busy function. Upon return from the APM Installation Check call, bit 2 of the CX register indicates if the system BIOS slowed the CPU clock rate during the CPU Idle call. You can use this bit to determine whether to call CPU Busy before executing code that should run at full speed.

Calling CPU Busy is discouraged when the system is already operating at full speed because of the unnecessary call overhead, but the operation is allowed and has no unexpected side effects.

Call With: AH = 53H

AL = 06H

Return Value: CY = 00H

Notes: This function is available in both the APM INT 15H and protect mode interface. Either APM or IPM busy can keep the CPU processing at full speed.

CPU Idle

Purpose: This function informs the system BIOS that the system is currently idle and that processing can be suspended until the next system event (typically an interrupt) occurs. This function allows the system BIOS to take some implementation-specific power saving action, such as stopping the CPU clock.

In cases where an interrupt causes the system to leave the idle state, the interrupt might have been serviced when the BIOS returns from the CPU Idle call. The calling application should not make any assumptions concerning interrupt servicing and should allow pending interrupts to be taken upon return from CPU Idle.

If interrupts are serviced from within the BIOS CPU Idle function, the interrupt handler must return to the BIOS when interrupt processing is completed. The caller cannot use knowledge of being in the idle state to regain control from an interrupt handler. For example, some system implementations can slow the processor CPU clock rate before waiting on an interrupt, and restore the normal clock rate after the interrupt is serviced but before returning from the idle call.

When the caller regains control, it should determine if there is actually any processing waiting to be performed and reissue the CPU Idle call if not. If the caller is a multitasking supervisor, it may be necessary to dispatch all of the applications (through calls to the supervisor), allowing them to check for incomplete activity.

Call With: AH = 53H

AL = 05H

Return Value: CY = 00H

Notes: This function is available in both the APM INT 15H and protect mode interface. Either the APM or IPM interface can keep the CPU in the busy state.

Enable/Disable Power Management

Purpose: This function enables or disables all APM automatic power down functionality. When disabled, the system BIOS does not automatically power down devices, enter the Standby state or Suspend state, or take power-saving steps in response to CPU Idle calls. In addition, many system BIOS functions are disabled and return the error 01H, indicating that power management functionality is disabled.

Call With:

AH = 53H

AL = 08H

BX = FFFFH Enable / disable all power management

CX = 00H Disable power management
01H Enable power management

Return Value: If the function is successful:

Carry flag = Clear

If the function is unsuccessful:

Carry flag = Set

AH = Error code:
01H Power management functionality disabled
09H Unrecognized device ID
0AH Parameter value in CX is out of range

Notes: This call is available in both the APM INT 15H and protect mode interfaces. Either APM or IPM can use this call to disable power management. Both APM and IPM must be enabled before enabling power management.

Get PM Event

Purpose: This function returns the next pending PM event (or indicates that no PM events are pending).

Call With: AH = 53H

AL = 0BH

Return Value: If the function is successful:

Carry flag = Clear

BX = PM event code:
- 01H System Standby request notification
- 02H System Suspend request notification
- 03H Normal resume system notification
- 04H Critical resume system notification
- 05H Battery low notification

If the function is unsuccessful:

Carry flag = Set

AH = Error code:
- 03H Interface connection is not established
- 80H No PM events are pending

Notes: Power events can occur when the system software is not immediately ready to process the event. In order to synchronize power events with other system software, the system BIOS withholds event notification until polled with the get PM event function. The cooperative PM software is only notified when it is ready to process the event.

The PM partner receives power event notification by periodically calling the get PM event function. Since power management events are not extremely time critical, a polling frequency of once or twice a second should be sufficient.

The 01H system Standby request notification informs the PM partner that the system BIOS wants to place the system in a Standby state. The system BIOS does not take any action at this time, but waits passively for the PM partner to call the system Standby function.

The 02H system Suspend request notification informs the PM partner that the system BIOS wants to place the system into a Suspend state. The system BIOS does not actually perform a Suspend at this time, but waits passively for the partner to call the Suspend system function.

A Resume system notification indicates that a critical system Suspend and Resume operation occurred without the cooperation of the PM partner. When a Suspend operation is scheduled to occur, the system BIOS normally notifies the partner through the Suspend system request, but does not actually perform the Suspend until the partner calls the Suspend system function.

In some emergency situations the system BIOS needs to Suspend the system immediately. In this situation, the system BIOS informs the partner that the system has resumed from such a Suspend through a Resume system notification. The partner must recover from the Suspend as best it can. Critical Suspend situations include power on and off, low battery, dead battery, and battery interlock.

The 05H battery low notification informs the PM partner that the system battery is running low.

This call is available in both the APM INT 15H and protect mode interfaces.

Get Power Status

Purpose: This function returns the system current power status.

Call With: AH = 53H

AL = 0AH

BX = 0001H

Return Value: If the function is successful:

Carry flag = Clear

BH = AC line status:
00H 0 indicates offline
01H 1 indicates online
FFH 255 indicates unknown
All other values reserved

BL = Battery status:
00H 0 indicates high
01H 1 indicates low
02H 2 indicates critical
03H 3 indicates charging
FFH 255 indicates unknown
All other values reserved

CL = Remaining battery life:
00H–64H Percent of full charge (0–100) in increments of 10
FFH 255 indicates unknown

If the function is unsuccessful:

Carry flag = Set

AH = 09H Unrecognized device ID

Notes: This call is available in both the APM INT 15H and protect mode interfaces.

Interface Connect

Purpose: This function establishes the cooperative interface between the caller and the system BIOS. Before the interface is established, the system BIOS provides its own power management functionality as implemented by the OEM. Once the interface is connected, the system BIOS and the caller coordinate PM activities according to this specification.

Call With:

AH = 53H

AL = 01H

BX = 0000H System BIOS

Return Value: If the function is successful:

Carry flag = Clear

If the function is unsuccessful:

Carry flag = Set

AH = Error code:
- 02H Interface connection is already in effect
- 09H Unrecognized device ID

Notes: This call is only available using the APM INT 15H interface.

Interface Disconnect

Purpose: This function breaks the cooperative interaction between the system BIOS and the caller and in the process restores the system BIOS default functionality. Any protected mode connection set up by the protected mode 16-bit interface connect or protected mode 32-bit interface connect functions is also invalidated by this call.

Even though this call returns control of power management strategy to the system BIOS, the parameter values (such as timer values and enable and disable settings) in effect at the time of the disconnect remain in effect.

Call With: AH = 53H

AL = 04H

BX = 0000H System BIOS

Return Value: If the function is successful:

Carry flag = Clear

If the function is unsuccessful:

Carry flag = Set

AH = Error code:
- 03H Interface is not connected
- 09H Unrecognized device ID

Notes: This call is available in both the APM INT 15H and the protect mode interfaces.

Protected Mode 16-Bit Interface Connect

Purpose: This function initializes the optional 16-bit protected mode interface between the caller and the system BIOS. This interface allows a protected mode caller to invoke the system BIOS functions without the need to first switch into real or virtual-86 mode. A caller that does not operate in protected mode might not need to use this call. This function establishes a 16-bit protected mode interface, but you must invoke this function in either real or virtual-86 mode using the INT 15H interface.

Call With:

AH = 53H

AL = 02H

BX = 0000H System BIOS

Return Value: If the function is successful:

Carry flag = Clear

AX = PM 16-bit code segment (real mode segment base address)

BX = Offset of entry point

CX = PM 16-bit data segment (real mode segment base address)

If the function is unsuccessful:

Carry flag = Set

AH = Error code:

- 05H 16-bit protected mode interface already established
- 06H 16-bit protected mode interface not supported
- 09H Unrecognized device ID

Notes: The system BIOS 16-bit protected mode interface requires two consecutive selector and segment descriptors to use for a 16-bit code and data segment, respectively. The caller must initialize these descriptors using the segment base information returned by the BIOS.

The segment limit fields are set to 64K. These selectors can either be in the Global Descriptor Table (GDT) or Local Descriptor Table (LDT), but must be valid whenever the system BIOS is called in protected mode. The code segment descriptor must specify protection level 0 and must invoke the BIOS routines with CPL = 0 so they can execute privileged instructions.

The caller invokes the system BIOS using the 16-bit interface by making a far call to the initialized code segment selector with the offset returned in BX. The caller must supply a stack large enough for the BIOS and potential interrupt handlers to use. The caller's stack is active when interrupts are enabled in the BIOS routine; the BIOS does not switch stacks when interrupts are enabled, including nonmaskable interrupts (NMI). The BIOS 16-bit interface requires a call using a 16-bit stack.

When you call the system BIOS routines in protected mode, the current I/O permission bit map must permit access to any I/O ports that the BIOS might need.

The system BIOS rejects additional 16-bit protected mode connections when a connection has already been established.

This call is only available using the APM INT 15H interface.

Protected Mode 32-Bit Interface Connect

Purpose: This function initializes the optional 32-bit protected mode interface between the caller and the system BIOS. This interface invokes the system BIOS functions without needing to switch into real or virtual-86 mode first. A caller that does not operate in protected mode might not need to use this call. This function establishes a 32-bit protected mode interface, but must be invoked in either real or virtual-86 mode using the INT 15H interface.

Call With: AH = 53H

AL = 03H

BX = 0000H System BIOS

Return Value: If the function is successful:

Carry flag = Clear

AX = PM 32-bit code segment (real mode segment base address)

BX = Offset of entry point

CX = PM 16-bit code segment (real mode segment base address)

DX = PM data segment (real mode segment base address)

If the function is unsuccessful:

Carry flag = Set

AH = Error code:
- 07H 32-bit protected mode interface already established
- 08H 32-bit protected mode interface not supported
- 09H Unrecognized device ID

Notes:

The system BIOS 32-bit protected mode interface requires three consecutive selector/segment descriptors for use as 16- and 32-bit code and data segments, respectively. The system BIOS 32-bit interface needs both 16- and 32-bit code descriptors to call other BIOS routines in a 16-bit code segment if it becomes necessary. The caller must initialize these descriptors using the segment base information returned in the call from the BIOS.

The segment limit fields are set to 64K. These selectors can be either in the GDT or LDT, but must be valid whenever the system BIOS is called in protected mode. The code segment descriptors must specify protection level 0 and must invoke the BIOS routines with CPL = 0 so that they can execute privileged instructions.

The caller invokes the system BIOS using the 32-bit interface by making a far call to the initialized 32-bit code segment selector with the offset returned in BX. The caller must supply a stack large enough for the BIOS and potential interrupt handlers to use. The caller stack is active when interrupts are enabled in the BIOS routine; the BIOS does not switch stacks when interrupts are enabled, including NMI interrupts. The BIOS 32-bit entry point requires a call using a 32-bit stack.

When you call the system BIOS routines in protected mode, the current I/O permission bit map must permit access to the I/O ports the BIOS might need.

The system BIOS rejects additional 32-bit protected mode connections if a 32-bit protected mode connection has already been established.

This call is only available using the APM INT 15H interface.

Restore System BIOS Power-On Defaults

Purpose: This function directs the system BIOS to reinitialize all its power-on defaults.

Call With: AH = 53H

AL = 09H

BX = FFFFH

Return Value: If the function is successful:

Carry flag = Clear

If the function is unsuccessful:

Carry flag = Set

AH = Error code:
09H Unrecognized device ID

Notes: This call is available in both the APM INT 15H and protect mode interfaces.

Set Power State

Purpose: This function places the system or device specified in the power device ID into the requested state.

Call With: AH = 53H

AL = 07H

BX = Power device ID

CX = System state ID
0000H Ready (Not supported for system device ID 0001H)
0001H Standby
0002H Suspend
0003H Off (Not supported for system device ID 0001H)
0004H–FFFFH Reserved

Return Value: If the function is successful:

Carry flag = Clear

If the function is unsuccessful:

Carry flag = Set

AH = Error code:
01H Power management functionality is disabled
09H Unrecognized device ID
0AH Parameter value in CX is out of range
60H Cannot enter the requested state

Notes: This call is available in both the APM INT 15H and protect mode interfaces.

Suspend System

Purpose: This function directs the system BIOS to place the system into the low-power Suspend state. The caller invokes this function in response to a Suspend system request notification from the system BIOS. First, the caller finishes any incomplete processing of its own, and then the caller invokes the Suspend system interrupt.

In addition to responding to system BIOS requests, the caller can initiate a Suspend operation. For example, if the caller determines that the system has been completely idle during a given time interval (for other than normal background processing such as servicing timer interrupts), it could request a system Suspend using this function.

Because the system BIOS does not return to the caller until the system is resumed from the Suspend state, there is no need to check for most resume notifications from the system BIOS. Upon return, and when safe to do so, the caller can notify any PM-aware applications of the resume status.

Since the system may have been suspended for a long period of time, the caller might need to update the date and time values.

Call With: AH = 53H

AL = 07H

BX = 0001H

CX = 0002H

Notes: This call is available in both the APM INT 15H and protect mode interfaces.

System Standby

Purpose: This function commands the system BIOS to place the system into the Standby state. The caller invokes this function in response to a system standby request notification from the system BIOS after the caller has performed any processing of its own. The caller can also initiate a standby operation.

For example, if the caller determines that the system is idle and is not already in the Standby state, it could request a system standby using this function.

Typically, you exit the Standby state by raising an interrupt. Interrupts are serviced normally.

Call With: AH = 53H

AL = 07H

BX = 0001H

CX = 0001H

Notes: This call is available in both the APM INT 15H and protect mode interfaces.

5

Using the PSK With DCM

The Programmer's Software Kit Language Libray includes three C language functions you can use for accessing a remote database from the JANUS reader.

Accessing a Database Server Through DCM

You can easily write an application program to run on the JANUS reader that accesses a remote database server through Intermec data collection manager (DCM) software. DCM does not run on the reader. It runs on a separate PC or workstation and provides transparent access to the remote database. The database may or may not be on the same computer as DCM.

You need the following items to access a database server:

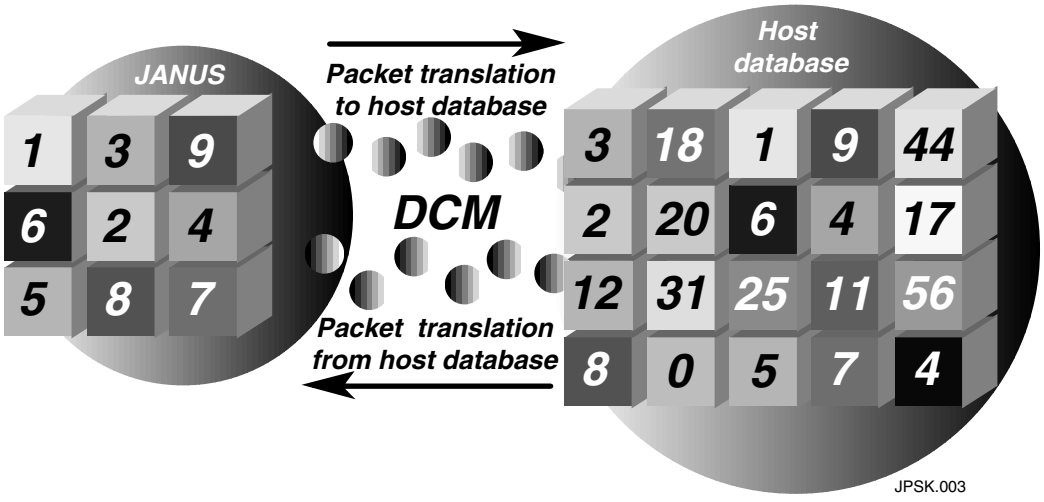
- The database table definition, including field types and field lengths
- Intermec DCM software version 4.3 or higher configured to access the database
- im_setup_trx, which sets up the protocol parameters for accessing the database server
- im_standard_trx, which packs and sends a transaction to the database
- im_parse_host_response, which reads the response from the host

Note: *These functions are only available in the C language library*.

Defining a Database Transaction

DCM handles the actual translation of a packet of data from the reader into a database-usable packet, based on settings that you configure in DCM. Your application defines the variables for the data, calls the function to pack and transmit the data, and handles the response from the database.

You must understand how the data on the reader maps to the data in the database structure and design your application and the DCM configuration around this relationship. The following figure shows how the information in the reader relates to the information in the database.



For this chapter, the term *transaction* means a function call on the JANUS reader that sends a message to the database server and then receives and distributes the returned values. The transaction includes a transaction ID, data fields, and a database operation request. The database operation can be a read, a write, an update, or a delete. The return message is ACK, NAK, or data.

A read operation returns a status code and record information from the host that your application can process with im_parse_host_response (if pacing is off). The other database operations return only status codes.

To set up a reader to database transaction

1. Configure the database destination in DCM and map the fields in the JANUS reader to the database table.
2. Send a transaction to the database using im_standard_trx.
3. Check the return status codes from im_standard_trx.
4. If the transaction is a read operation, parse the returned records until the JANUS reader receives an ACK.
5. If needed, provide a standby file for transmitting transactions later if the transmit channel is unavailable.

The following sections explain each step.

Configuring DCM

Before writing your application on the JANUS reader, you need to configure DCM for the database transactions. Refer to the appropriate DCM user's manual for the database actions available and for detailed instructions on configuring DCM.

Manual	Part Number
DCM for OS/2 Manual Kit	057433
DCM for HP-UX Manual Kit	062223
DCM for SCO Manual Kit	062765

To set up DCM to access a database

1. Define the database interface and network configuration.
2. Define the database action: read, insert (write), update, or delete.
3. Define the JANUS reader transaction fields and map the fields to the database table. For example, the second field in the JANUS reader transaction is numeric and corresponds to the third column in the database table.

Sending a Transaction to the Database

You use im_setup_trx and im_standard_trx for defining protocol parameters and sending a transaction to a database. You can use the default im_setup_trx protocol values, without calling im_setup_trx, or you can define your own values. The following table lists the default protocol values. For more information, see Chapter 2, "C Language Support."

Description	Default Value
Communications port (COM1)	IM_COM1
Transmit timeout value (infinite)	IM_INFINITE_TIMEOUT
Receive timeout value (infinite)	IM_INFINITE_TIMEOUT
Read one record at a time (only affects read operations)	IM_DBPACE_ENABLE
Standby file and batch processing disabled	im_batch_enable is off
Do not ignore NAK during batch processing	im_dobest_enable is off
System delimiter character (comma)	,
Field delimiter character (comma)	,
Maximum packet length	256 characters
No standby file (*safety* = 0)	0

Note: *Handshaking is always on when the JANUS reader accesses a host database.*

Your application handles two cases of database actions: database reads and other database transactions. A read operation returns data that your application must process. The data is either an ACK or NAK mesage from the database host or data fields that must be parsed. Database write, update, and delete operations return only an ACK or NAK message from the database host.

To set up a database write, update, or delete

1. If needed, use im_setup_trx to set up the protocol for the database server.
2. Use im_standard_trx to pack and transmit a record to the database.
3. Provide error trapping by checking the return status codes.

To set up a database read

1. If needed, use im_setup_trx to set up the protocol for the database server.
2. Use im_standard_trx to pack and transmit a record to the database.
3. Provide error trapping by checking the return status codes.
4. Use im_parse_host_response to parse the returned records until the end of the database records is reached.

Checking the Return Status Codes

The database functions provide many return codes to help you pinpoint an error. You must decide which errors to check for and how your application will handle each situation. In some situations your application may only need to verify the category of the error. In others, your application needs to verify the exact error condition.

For example, a database read operation may successfully read the database, but the data returned to the JANUS reader may overflow the receive buffer you defined. Your application can use one of the status code macros defined in Chapter 2, “C Language Support.” You can also test for the successful category by testing bit 4 in the return code. If bit 4 is set to 1 (mask with 0x0010), you know that the database operated correctly and sent at least one record to the JANUS reader. However, you do not know that all of the data was received.

If your application needs to verify the exact error and provide a correction, you need to test for the specific status value. The following example shows two ways to test the return codes.

Example

```
*  Checking return codes with macros and by value
*
status = im_standard_trx(pszTid, IM_DBRQT_DATA, pszRead[i], 256,
                         pszInString, pszGetData, &chOperation);
if im_isgood(status)     //See if any data has been received.
{
                         //Process received data.
                         //Add your code here.
}

*  Test for receive data and specific return code
status = im_standard_trx(pszTid, IM_DBRQT_DATA, pszRead[i], 256,
                         pszInString, pszGetData, &chOperation);
if (status == IM_DCM_ACK_RCD)
{
                         //Received data and successful parse.
                         //Add your code here.
}
   elseif (status == IM_DCM_TRUNCATED)
{
                         //Received data but parse buffer too small
                         //Add your code here.
}
    elseif (status == IM_DCM_EXCESSFIELD)
{
                         //Received data but too many fields
                         //Add your code here.
}
```

The example referred to the symbolic name of the return code, such as IM_DCM_ACK_RCD. You can also check for the exact hex value. The following sections list the return code hex values for the DCM functions im_parse_host_response, im_setup_trx, and im_standard_trx, grouped as follows:

- By calling parameter
- In numerical order
- Symbolic return code with its hex value

Return Codes Listed by Calling Parameter

The following two tables list the return codes arranged by the calling parameters and severity. The first table lists the return codes for the parameters that affect the standby file. The second table lists the return codes for accessing the remote database.

im_standard_trx Return Codes for Manipulating the Standby File

Request Type Parameter	Return Code	Description
IM_ERROR_RECORD	1A00	Retrieved record from standby file
	5A07	Standby file is empty
	9A08	Standby file I/O error
	9A09	Standby file is missing
IM_ADVANCE_RECORD	1A00	Retrieved record from standby file
	5A07	Standby file is empty
	9A08	Standby file I/O error
	9A09	Standby file is missing
IM_ZIP_RECORD	1A00	All standby records sent successfully
	9A09	Standby file is missing
	5A02	NAK received. If im_dobest_enable is not set, then stop sending records.
	9A03	Transmit channel error
	9A04	Receive channel timeout
	9A05	Packet header error
	9A06	Missing transaction ID

im_standard_trx Return Codes for Accessing the Remote Database

Request Type Parameter	Return Code	Description
All	1A00	Transaction successful
	1A10	Received message on Read operation
	5A11	Data was truncated
	5A12	Too many fields
	1A01	Transaction successful, ACK
	5A02	Transaction failed, NAK
	5A20	Record stored in standby file successfully
	5A21	Record stored in standby file, but upload standby error
	9A03	Transmit failed
	9A04	Receive port timeout
	9A40	Buffer overflow
	9A41	Buffer overflow with upload standby error
	9A05	Packet header error
	9A06	Missing transaction ID
	9A08	Standby file I/O error
	9A09	Standby file is missing

im_parse_host_response Return Codes

Request Type Parameter	Return Code	Description
All	1A10	Received message
	5A11	Data was truncated
	5A12	Too many fields
	1A01	Transaction success, ACK
	5A02	Transaction failed, NAK
	9A04	Receive port timeout
	9A05	Packet header error

im_setup_trx Return Codes

Request Type Parameter	Return Code	Description
All	1A00	Protocol parameters set successfully
	9A08	Standby file I/O error, check system errno value
		Standby file format is wrong

Return Codes Listed in Numerical Order

The following tables list the return codes for each DCM function in numerical order.

im_standard_trx Return Codes

Return code	Severity	Description
1A00	00 Success	Action successful
1A01	00 Success	Transaction successful, ACK received
1A10	00 Success	Received message on Read operation
5A02	01 Warning	Transaction failed, NAK received. If im_dobest_enable is not set, then stop sending records.
5A07	01 Warning	Standby file is empty
5A11	01 Warning	Data was truncated
5A12	01 Warning	Too many fields
5A20	01 Warning	Record stored in standby file successfully
5A21	01 Warning	Record stored in standby file, but upload standby error
9A03	10 Error	Transmit failed
9A04	10 Error	Receive port timeout
9A05	10 Error	Packet header error
9A06	10 Error	Missing transaction ID
9A08	10 Error	Standby file I/O error
9A09	10 Error	Standby file is missing
9A40	10 Error	Buffer overflow
9A41	10 Error	Buffer overflow with upload standby error

im_parse_host_response Return Codes

Return code	Severity	Description
1A01	00 Success	Transaction success, ACK
1A10	00 Success	Received message
5A02	01 Warning	Transaction failed, NAK
5A11	01 Warning	Data was truncated
5A12	01 Warning	Too many fields
9A04	10 Error	Receive port timeout
9A05	10 Error	Packet header error

im_setup_trx Return Codes

Return code	Severity	Description
1A00	00 Success	Protocol parameters set successfully
9A08	10 Error	Standby file I/O error, check system errno value
		Standby file format is wrong

Symbolic Return Codes

This table lists the return codes for all three DCM functions.

Return Code	Hex Value	Description
IM_DCM_OK	1A00	Execution success
IM_DCM_ACK_RCD	1A01	ACK received
IM_DCM_MSG_RCD	1A10	Data received
IM_DCM_NAK_RCD	5A02	NAK received
IM_DCM_NO_RECORD	5A07	No more log records
IM_DCM_TRUNCATED	5A11	Received data truncated
IM_DCM_EXCESSFIELD	5A12	Fields overflow
IM_DCM_STANDBY	5A20	Record logged
IM_DCM_STANDBY_TXERR	5A21	Logged but upload error
IM_DCM_NOTRANSMIT	9A03	Transmit fails
IM_DCM_TIMEOUT	9A04	Receive timeout
IM_DCM_HEAD_ERR	9A05	Packet header error
IM_DCM_TRANSID_ERR	9A06	Transaction ID is missing
IM_DCM_NO_HANDLE	9A08	Standby file is missing
IM_DCM_FILE_ERR	9A09	Standby file I/O error
IM_DCM_FILLED	9A40	Buffer overflow
IM_DCM_FILLED_TXERR	9A41	Buffer overflow and upload error

Using a Standby File

The im_setup_trx function supports using a standby file to store records when the transmit channel is unavailable. For example, you may use the JANUS reader to collect inventory data from a warehouse. The reader stores the records until you set it in the communications dock and then transmits the standby records to a database. By default, the standby file is not activated, so your application program must handle activating and deactivating the standby file.

You activate the standby file by opening a file and passing the file handle as the *hsafety* parameter of im_setup_trx. You deactivate the standby file by passing 0 for the *hsafety* parameter. For more information on the im_setup_trx parameters, see Chapter 2, "C Language Support."

Your application must open and close the standby file in low level file I/O open() with binary mode and readable/writeable mode. For example, you may open a file named standby.txt as follows:

```
handle = open("standby.txt", O_CREAT | O_BINARY | O_RDWR,
S_IREAD | S_IWRITE);
```

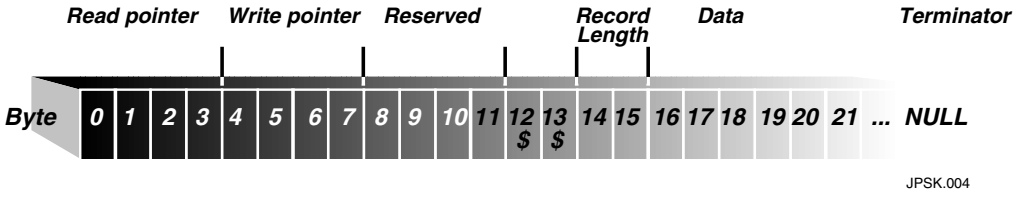
When you pass the standby file handle the first time after creating the file, im_setup_trx initializes the standby file with the correct format for you.

Standby File Structure

This section provides detailed file structure information for troubleshooting. Usually you do not need this level of detail. The PSK library functions and the following open statement create the correct file.

```
handle = open("standby.txt", O_CREAT | O_BINARY | O_RDWR,
S_IREAD | S_IWRITE);
```

The first 14 bytes of the standby file are the file header. The remainder of the file contains records. The following figure shows the header structure. The first 4 bytes of header is the read pointer and the next 4 bytes of header is the write pointer. The next 4 bytes are reserved for future use. The last 2 bytes of the header are two $ characters that identify this file as a standby file.



Each record starts with 2 bytes for length of the record. Each record is variable length and ends with a null character.

If you activate the standby file and the file length is zero, the standby file is initialized with a header but no records. If the file length is not zero (0), the file header is read.

Each record logged to the standby file is written at the byte specified by the write pointer. After the new record is written, the write pointer in the header is updated.

When the transmit channel is available again, the logged records are uploaded. As each record is read for uploading, the read pointer in the header is updated. When all the records have been sent, the read pointer reaches the write pointer. Then the file is reset to the initial state, which has only the 14 byte header in the file.

The standby file retains data between program runs. You can exit your application or reboot the reader without losing data. However, if you reboot the reader while an application is running, the standby file may contain corrupt data. In this case you need to delete the standby file so that the next application will run correctly.

To simplify error trapping, you do not need to use a standby file when you first design an application that uses DCM's database server. After you test the application design for the database server protocol and accessing the database, then put the standby file function into the program. In this way, you can avoid the confusion of deciphering too many error return codes.

Sample Program

```
//********************im_setup_trx    ***************************************
//********************im_standard_trx ***************************************
//********************im_parse_host_response ********************************
// This program demonstrates the PSK's DCM support functions.
//
// First, it discards any records left in the database server queue by a
// previous read transaction.
// Second, it deletes all records from the BONUS table.
// Third, it uses the batch capability to store 5 records in the
// standby file, and then write the 5 records to the database.
// Fourth, it writes the remaining records in the reader without batching them.
// Fifth, it reads records from the database without pacing and then with pacing.
// Sixth, it demostrates how to discard records after reading the first
// 3 records.
//
// The program is divided into the following routines:
//  DemoInit()    - Initializes the database server
//  DemoBatch()   - Shows the batch process case
//  DemoStandby() - Shows the standby file capability
//  HandNoPace()  - Reads records with handshaking and no pacing protocol
//  HandPace()    - Reads records with handshaking and pacing protocol
//  HandDiscard() - Reads 3 records and then discards the rest of records.
//
// Note: You must load reader services and a protocol handler on the JANUS reader
//       to enable send/receive buffer functions.
//       Run these TSR's at DOS prompt before running this program:
//         rservice
//         phimec 1  ( 1 is COM1)
//       On the DCM side, you need to set up database table access and the TSTDEL,
//       TSTINS, TSTRD transactions.
//*****************************************************************************

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <dos.h>
#include <string.h>
#include <fcntl.h>
```

```
#include <io.h>
#include <sys\stat.h>
#include "imgenric.h"
#include "im20lib.h"


typedef  struct BonusType {
char  *emp;                    //Mapped to BONUS table's NAME column
char  *dept;                   //Mapped to BONUS table's DEPT column
char  *salgrade;               //Mapped to BONUS table's SALGRADE column
char  *end;
} BONUSTYPE;

static BONUSTYPE EmpTable[] ={
 {"Steve Chan",     "Network",  "1",  NULL},
 {"Stuart Bainey",  "Software", "2",  NULL},
 {"Trixie Austin",   "Hardware", "3",  NULL},
 {"Marcus Albrod",   "Portable", "4",  NULL},
 {"Jeff Eastone",     "Network",  "5",  NULL},
 {"Roy Christoff",    "Software", "6",  NULL},
 {"KK Luang",   "Hardware", "7",  NULL},
 {"Joe Arant",    "Portable", "8",  NULL},
 {"Matt Hartinger", "Network",  "9",  NULL},
 {"Tom Dard",       "Software", "10", NULL},
 {"Ted Millen",      "Hardware", "11", NULL},
 {"Robert Chian",    "Portable", "12", NULL},
 {"Doug Kaatz",     "Network",  "13", NULL},
 {"RuthAnn Hso",       "Software", "14", NULL},
 {"Will Powers",    "Hardware", "15", NULL},
 {"Kevin Cuffman",    "Portable", "16", NULL},
 {"C. K. Koonze",   "Network",  "17", NULL},
 {"Ray Law",        "Software", "18", NULL},
 {"Mark Robinns",   "Hardware", "19", NULL},
 {"Cathy McCree", "Portable", "20", NULL},
 {"Shar Leung",    "Network",  "21", NULL},
 {"Jack Lincoln",       "Software", "22", NULL},
 {"Joe Schoening",    "Hardware", "23", NULL},
 {"Bill McDowell",  "Portable", "24", NULL},
 {"Sophia Renouard", "Network",  "25", NULL},
 {"Yong Qin Lee",    "Software", "26", NULL},
 {"Joe Waters",       "Hardware", "27", NULL},
 {"Paul Swann",   "Portable", "28", NULL},
 {"Mack Pickett",   "Network",  "29", NULL},
 {"Raymond Wallice",     "Software", "30", NULL},
 {"Delanie Zerbi",  "Hardware", "31", NULL},
 {"Lisa Vanian", "Portable", "32", NULL},
 {NULL,             NULL,       NULL, NULL}
};


static IM_COM_PORT  iPort = IM_COM1;    /* Uses communications port 1*/

IM_UCHAR    chOperation = 'R';          //'W', ' Operation char


#define  FLDSIZE     16
static char pszEmployee[FLDSIZE+1]="\0"; //Buffer mapped to Employee
static char pszDepart[FLDSIZE+1]="\0";   //Buffer mapped to department
static char pszSalary[FLDSIZE+1]="\0";   //Buffer mapped to Salary grade
static char pszDummy[FLDSIZE+1]="\0";    //Buffer mapped to dummy
```

```
//data structure to get fields from packet.
static IM_DCMSTRN   pszGetData[] = {
  { pszEmployee, sizeof(pszEmployee)},
  { pszDepart,   sizeof(pszDepart)},
  { pszSalary,   sizeof(pszSalary)},
  { pszDummy,    sizeof(pszDummy)},
  { NULL, 0}
};


#define DEPT_NUM    4             //Number of dempartments

static char *pszReadSoftware[] ={"Software", NULL};
static char *pszReadHardware[] ={"Hardware", NULL};
static char *pszReadNetwork[]  ={"Network",  NULL};
static char *pszReadPortable[] ={"Portable", NULL};

static char  **pszRead[DEPT_NUM + 1] = {
  pszReadNetwork,
  pszReadSoftware,
  pszReadHardware,
  pszReadPortable,
  NULL
};

static IM_UCHAR     pszInString[257];      // pointer to output string

static char         standby[]="standby.bin";      //Standby file name
static int          standby_fh = 0;

void DemoInit(void );
void DemoBatch(void );
void DemoStandby(void );
void HandNoPace(void);
void HandPace(void);
void HandDiscard(void);

/*************************************************************************/

void main (void)
{


  //Always open standby file for use
  if ( (standby_fh = open(standby, O_CREAT | O_BINARY |O_RDWR, S_IREAD|S_IWRITE)) == -1)
   {
       printf("open file error\n");
       exit(1);
   }

   //Send IM_DBRQT_DISCARD to discard all records left over from database server
   DemoInit();

   //Use IM_BATCH_ENABLE to delete all records in BONUS table
   DemoBatch();

   //Use standby function to insert records to BONUS table
   DemoStandby();

   //Using handshaking with no pacing protocol
   //Read records that match dept criteria, using a FOR loop without pacing
   HandNoPace();
```

```
   //Using handshaking with pacing protocol
   //Read records that match dept criteria, using a FOR loop with pacing
   HandPace();

   //Using handshaking with pacing protocol
   //Read 3 records that match dept criteria, and then discard the rest of records
   HandDiscard();

}

//*********************************************************************
// Initialize database access.
//  First, read data from communications port until timeout occurs, which
//  means no more data received.
//  Use TSTRD with IM_DBRQT_DISCARD to discard any records left in
//  the database.
//*********************************************************************

void DemoInit(void)
{
IM_USHORT    status;
IM_USHORT    actual_length = 0;
IM_USHORT    iDBPrtcl;              // handshake, pace, request type
IM_UCHAR     com_buff[257];         // Buffer for receiving


   //Flush out all packets left from previous session.
   im_cancel_rx_buffer(iPort);    //Clear buffer
   //Read data until timeout occurs

   do
   {
      status = im_receive_buffer(iPort, sizeof(com_buff)-1, com_buff,
                                 3000, &actual_length);
      if (status == 0x0600 && actual_length > 0)
        printf("%s\n", com_buff);

   } while ( status != 0x4601);



   iDBPrtcl = IM_DBPACE_ENABLE;
   //Set up communications port and protocol
   status = im_setup_trx(iPort, 240, IM_INFINITE_TIMEOUT, iDBPrtcl,
                         ',', ',', 256,  standby_fh);

   if (status != IM_DCM_OK)
        printf("error: %x\n", status);

   do
   {
      //Send discard transaction to the database server.
      status = im_standard_trx(RedpszTid, IM_DBRQT_DISCARD, NULL, 256,
                               pszInString, pszGetData, &chOperation);

      if (status != IM_DCM_ACK_RCD && status != IM_DCM_NAK_RCD )
         printf("Init error: %x\n", status);

   } while ( status != IM_DCM_NAK_RCD && status != IM_DCM_ACK_RCD);

   printf("\nInit Complete!\n");
```

```
   //Wait for 1 second (optional)
   im_standby_wait(1000);

}

//*********************************************************************
//Demostrate batch process to clean up the database table.
//  Log records into the standby file, and then flush them.
//  Use TSTDEL to delete all records from BONUS table.
//  Use IM_BATCH_ENABLE to log transactions, then process in batch mode
//*********************************************************************

void DemoBatch(void)
{
BONUSTYPE    *pszBonus;
IM_USHORT    status;
IM_USHORT    del_status;
IM_USHORT    iDBPrtcl;               // handshake, pace, request type
IM_UCHAR     pszTid[8]="TSTDEL";     //delete transaction


   //Turn on batch process
   iDBPrtcl = IM_BATCH_ENABLE + IM_DBPACE_ENABLE;

   status = im_setup_trx(iPort, 240, IM_INFINITE_TIMEOUT, iDBPrtcl,
                         ',', NULL, ',', 256,  standby_fh);

   if (status != IM_DCM_OK)
      printf("Batch setup error: %x\n", status);


   //Send all records first
   pszBonus = EmpTable;

   while (pszBonus->emp != NULL)
   {
      status = im_standard_trx(pszTid, IM_DBRQT_DATA, pszBonus, 256,
                               pszInString, pszGetData, &chOperation);

      //It will receive IM_DCM_STANDBY, because IM_BATCH_ENABLE is on
      if (status != IM_DCM_STANDBY)
           printf("Standby error: %x\n", status);

      pszBonus++;    //go to next record

   }

   //Flush standby records
   do
   {
      //Continue flushing standby records until receive IM_DCM_OK
      status = im_standard_trx(pszTid, IM_ZIP_RECORD, pszBonus, 256, pszInString,
                               pszGetData, &chOperation);

      //If the TSTDEL transaction deletes a record in table, then reader receives
      //  ACK and im_standard_trx() continues sending logged records until the
      //  standby file is empty. When the file is empty, the function returns
      //  IM_DCM_OK.
      // If the database table has no records, then the DCM TSTDEL transaction sends
```

```
      //  NAK to the reader. im_standard_trx() returns IM_DCM_NAK_RCD and any records
      //  remain in the standby file. Use IM_ADVANCE_RECORD to discard any
      //  remaining records in the standby file.
      // If IM_DOBEST_ENABLE is on, then do not check for IM_DCM_NAK_RCD.
      //  im_standard_trx() treats NAK the same as ACK and uploads the next record.
      if (status == IM_DCM_NAK_RCD)
      {
         printf("NAK: %s\n", pszInString);   //See which record is

         //Skip this TSTDEL transaction because no records in database table.
         del_status = im_standard_trx(pszTid, IM_ADVANCE_RECORD, pszBonus, 256,
                                      pszInString,pszGetData, &chOperation);

         //If Error occurs, print it
         if (del_status != IM_DCM_OK)
              printf("%x\n", del_status);
      }
   }

   while (status != IM_DCM_OK);
      printf("\nDelete Complete!\n");

}

//*************************************************************************
//Demostrate standby functionality.
// Use TSTINS transaction to insert all records into the BONUS table
// using a standby file.
// Use batch process to log five transactions, then turn off batch process
// and send remaining transactions in normal mode. Upload all standby
// records before sending current record.
//*************************************************************************

void DemoStandby(void)
{
IM_USHORT    i;
BONUSTYPE    *pszBonus;
IM_USHORT    status;
IM_USHORT    iDBPrtcl;               // handshake, pace, request type
IM_UCHAR     pszTid[8]="TSTINS";     //delete transaction.


   //Turn on Batch process
   iDBPrtcl = IM_BATCH_ENABLE  + IM_DBPACE_ENABLE;

   status = im_setup_trx(iPort, 240, IM_INFINITE_TIMEOUT, iDBPrtcl,
                         ',',  ',', 256,  standby_fh);

   if (status != IM_DCM_OK)
       printf("error: %x\n", status);

   //Process 5 records in batch mode
   pszBonus = EmpTable;

   for (i= 0; i < 5; i++)
   {
      status = im_standard_trx(pszTid, IM_DBRQT_DATA, pszBonus, 256,
                               pszInString, pszGetData, &chOperation);

      if (status != IM_DCM_STANDBY)
           printf("Standby error: %x\n", status);
```

```
      pszBonus++;    //go to next records
   }

   //Turn off batch processing
   iDBPrtcl = IM_DBPACE_ENABLE;
   status = im_setup_trx(iPort, 240, IM_INFINITE_TIMEOUT, iDBPrtcl,
                         ',', NULL, ',', 256,  standby_fh);

   if (status != IM_DCM_OK)
        printf("error: %x\n", status);

   //Sending the rest of records.
   while (pszBonus->emp != NULL)
   {
      status = im_standard_trx(pszTid, IM_DBRQT_DATA, pszBonus, 256,
                               pszInString, pszGetData, &chOperation);

      if (status != IM_DCM_ACK_RCD)
           printf("Sending error: %x\n", status);

           pszBonus++;    //go to next record
   }

   //Flush standby records and make sure no records left in standby file
   do
   {
      status = im_standard_trx(pszTid, IM_ZIP_RECORD, pszBonus, 256,
                               pszInString, pszGetData, &chOperation);

      //See what is problem
      if (status != IM_DCM_OK)
           printf("Flush error: %x\n", status);

   }
   while (status != IM_DCM_OK);
      printf("\nInsert Complete!\n");

   //Wait for 5 seconds
   im_standby_wait(5000);

}

//*********************************************************************
// Demostrate reading records without pacing protocol.
//  TSTRD transaction reads records, without pacing protocol.
//  It reads records by department, so it needs to loop thru 4 times.
//*********************************************************************

void HandNoPace(void)
{
IM_USHORT    i;
IM_USHORT    status;
IM_USHORT    iDBPrtcl;           // Handshake, No pace, request type
IM_UCHAR     pszTid[16]="TSTRD";

   //Handshaking, no pacing
   iDBPrtcl = 0;

   //Set up protocol
   status = im_setup_trx(iPort, 240, IM_INFINITE_TIMEOUT, iDBPrtcl,
                         ',',  ',', 256,  standby_fh);
```

```
   if (status != IM_DCM_OK)
        printf("error: %x\n", status);


   //Read records with FOR loop. Each loop covers each department.
   for (i = 0; i < DEPT_NUM; i++)
   {
      status = im_standard_trx(pszTid, IM_DBRQT_DATA, pszRead[i], 256,
                               pszInString, pszGetData, &chOperation);

      if (status & 0x0010)     //See if data has been received
      {
         do
         {
            //Display Salary Grade, Employee name, department on screen.
            printf("%s, %s, %s\n", pszGetData[2].pszField,
                    pszGetData[0].pszField, pszGetData[1].pszField);

            //Get next record because protocol is no pace.
            status = im_parse_host_response( 256, pszInString, pszGetData,
                                             &chOperation);

         } while ((status & 0x0010) != 0); //End of records is ACK/NAK,
                                           // instead of data received.
      }

      //Returns ACK when no more records coming from database
      if (status != IM_DCM_ACK_RCD)
      {
         printf("NoPace: %x\n", status);
         printf("%s\n", pszInString);
      }

   }
   printf("\nRead No Pace Complete!\n");

}

//*********************************************************************
// Demonstrate reading records with pacing protocol on.
//  TSTRD transaction reads records with pacing protocol on.
//  It reads records by department, so it needs to loop thru 4 times.
//*********************************************************************

void HandPace(void)
{
IM_USHORT    i;
IM_USHORT    status;
IM_USHORT    iDBPrtcl;           // handshake, pace, request type
IM_UCHAR     pszTid[16]="TSTRD";


   //Handshaking and pacing.
   iDBPrtcl =  IM_DBPACE_ENABLE;

   //Set up protocol
   status = im_setup_trx(iPort, 240, IM_INFINITE_TIMEOUT, iDBPrtcl,
                         ',',  ',', 256,  standby_fh);

   if (status != IM_DCM_OK)
        printf("error: %x\n", status);
```

```
   //Read records with FOR loop. Each loop covers each department.
   for (i= 0; i < DEPT_NUM; i++)
   {
      //Read first record back if record exists
      status = im_standard_trx(pszTid, IM_DBRQT_DATA, pszRead[i], 256,
                               pszInString, pszGetData, &chOperation);

      if ( status & 0x0010)     //data received
      {
         do
         {
            //Display Salary Grade, Employee name, Department on screen.
            printf("%s, %s, %s\n", pszGetData[2].pszField,
                    pszGetData[0].pszField, pszGetData[1].pszField);

            //wait 2 seconds
            im_standby_wait(2000);

            //Ask for next records
            status = im_standard_trx(pszTid, IM_DBRQT_NEXTROW, NULL, 256,
                                     pszInString, pszGetData, &chOperation);

         } while ((status & 0x0010) != 0); //End of records is ACK/NAK,
                                           // instead of data received.
      }

      //Returns ACK when no more records coming from database
      if (status != IM_DCM_ACK_RCD)
      {
         printf("Pace: %x\n", status);
         printf("%s\n", pszInString);
      }

   }
   printf("\nRead Pace Complete!\n");

}

//*********************************************************************
//Demostrate discarding records with pacing protocol on.
//*********************************************************************

void HandDiscard(void)
{
IM_USHORT    i;
IM_USHORT    j;
IM_USHORT    status;
IM_USHORT    iDBPrtcl;           // handshake, pace, request type
IM_UCHAR     pszTid[16]="TSTRD";
IM_UCHAR     chPaceChar;


   //Handshaking and pacing.
   iDBPrtcl =  IM_DBPACE_ENABLE;

   //Set up protocol
   status = im_setup_trx(iPort, 240, IM_INFINITE_TIMEOUT, iDBPrtcl,
                         ',',  ',', 256,  standby_fh);

   if (status != IM_DCM_OK)
        printf("error: %x\n", status);
```

```
   //Read 3 records with FOR loop. Each loop covers a department.
   for (i= 0; i < DEPT_NUM; i++)
   {
      //Read first record in database if record exists
      status = im_standard_trx(pszTid, IM_DBRQT_DATA, pszRead[i], 256,
                               pszInString, pszGetData, &chOperation);

      //Initialize count so that after 3 records,  we can discard the
      // rest of the records
      j = 0;
      if ( status & 0x0010)     //data received
      {
         do
         {
            j++;

            //Display Salary Grade, Employee name, department on screen.
            printf("%s, %s, %s\n", pszGetData[2].pszField,
                    pszGetData[0].pszField, pszGetData[1].pszField);

             if (j > 2)
               chPaceChar = IM_DBRQT_DATA;         //'S' -Discard records
             else
               chPaceChar = IM_DBRQT_NEXTROW;      //'N' -next row

             //wait 2 seconds
             im_standby_wait(2000);

             //Ask for next record or discard rest of records
             status = im_standard_trx(pszTid, chPaceChar, NULL, 256,
                                      pszInString, pszGetData, &chOperation);

         } while ((status & 0x0010) != 0);  //End of records is ACK/NAK,
                                            // instead of data received.
      }

      if (status != IM_DCM_ACK_RCD)
      {
         printf("Discard: %x\n", status);
         printf("%s\n", pszInString);
      }
   }  //for

   printf("\nRead Discard Complete!\n");

}
```

A

Status Codes

This appendix lists the status code hex values and error messages. Use the tables in this appendix to look up the meaning of a specific status code.

Status Code Bit Values

Most of the PSK functions return status codes. You can use the codes to test for error conditions that your application will act upon.

How you test for the status codes depends on the complexity of the function returning the status and your application needs. In some cases, you may use the status code macros discussed in "Status Code Macros" in Chapter 2 to check the severity level. In other cases, you may want to check the exact status code value.

Each status code is a 16-bit word structured as follows:

15	14	13	12	10	9	8	7	6	5	4	3	2	1	0
s	s	f	f	f	f	f	m	m	m	m	m	m	m	m

JPSK.002

where:

s is severity code: 00 = Success
10 = Error
01 = Warning
11 = Fatal

f is the subsystem code, explained later in this appendix.

m is the message code that identifies the error or warning condition.

The status codes are IM_USHORT (unsigned short) values.

The tables in this appendix list the hex value of the message codes.

Status Codes Listed Numerically

The following table lists the return codes and the error message text provided by im_message and imMessage. For each message, there is an associated status return code, subsystem, and severity.

Some messages are abbreviated to fit the display and to save memory on the JANUS reader. The table includes an easy-to-read version of the abbreviated messages.

Status Codes and Error Messages Listed in Numerical Order

Hex Code	Severity	Subsystem	Reader Message and Description
0000	Good	NO	Request successful
0001	Good	NO	Prohibited
0100	Good	RS	Rservice request success Reader services successful
0103	Good	RS	Rservice resume success Reader services resume successful
0200	Good	CM	Request success Request successful
0201	Good	CM	Read End Of Record End of record reached
0202	Good	CM	Read End Of File End of file reached
0300	Good	SC	Scan success Scan successful
0400	Good	DC	Success Successful command
0401	Good	DC	Symbology already enabled Symbology is already enabled
0402	Good	DC	Symbology not enabled Symbology is not enabled
040A	Good	DC	Good decode Successful decode
040B	Good	DC	Intermediate row re-read Intermediate row reread

Status Codes and Error Messages Listed in Numerical Order (continued)

Hex Code	Severity	Subsystem	Reader Message and Description
040C	Good	DC	Intermediate decoded Intermediate raw decoded
0421	Good	DC	Default command symbology
0422	Good	DC	Mixed ASCII format
0500	Good	RW	RW success Reader Wedge successful
0505	Good	RW	No rdr cmds parsed No reader commands parsed
0506	Good	RW	Valid rdr cmds parsed Valid reader commands parsed
0509	Good	RW	Accumulating rdr cmd Accumulating reader command
050A	Good	RW	Rdr cmd override Reader command override
050B	Good	RW	Rdr cmd enter accum Reader command enter accumulation
050C	Good	RW	Rdr cmd exit accum Reader command exit accumulation
050D	Good	RW	Accumulate multi-read labels Accumulating multiread labels
0510	Good	RW	ENTER rdr cmd parsed An Enter reader command was parsed
0513	Good	RW	Protected field rdr cmd parsed Protected field was parsed
0517	Good	RW	Keycode label parsed Keycode label was parsed
0518	Good	RW	Isolated ENTER cmd parsed An isolated Enter command was parsed
0519	Good	RW	Good rdr cmd on exit accum A reader command was parsed successfully on exit from accumulation
051A	Good	RW	Good rdr cmd on ENTER A reader command was parsed successfully when Enter was parsed

Status Codes and Error Messages Listed in Numerical Order (continued)

Hex Code	Severity	Subsystem	Reader Message and Description
051C	Good	RW	Enter accum parsed An Enter Accumulate has been parsed when accumulating
051D	Good	RW	Exit multi-read A label terminating a multiple-read sequence was scanned
051E	Good	RW	Enter multi-read A label beginning a multiple-read sequence was scanned
0521	Good	RW	No comms status No communications status to report
0525	Good	RW	Rdr cmds parsed Reader commands forwarded to the application were parsed
0600	Good	CU	Comm success Communications operation success
0601	Good	CU	Comm Buff done Communications buffer done
0800	Good	PM	PM success Power management successful
0A00	Good	TM	Timer success Timer operation successful
0B00	Good	BP	Beep success Beeper call successful
0E00	Good	IM	String Extract Complete
0E01	Good	IM	String Append complete
0E02	Good	IM	Search string found
0E03	Good	IM	String Copy complete
0E04	Good	IM	Operation success Successful operation
0F00	Good	LG	Successful operation
1000	Good	KB	Expanded keypad success Expanded keyboard buffer successful
100A	Good	KB	Expanded buffer enabled
100B	Good	KB	Expanded buffer disabled
1100	Good	SS	System Config success System configuration successful

Status Codes and Error Messages Listed in Numerical Order (continued)

Hex Code	Severity	Subsystem	Reader Message and Description
1200	Good	KP	Keypad success Successful keypad operation
1300	Good	DP	Display success Successful display operation
1A00	Good	EX	Execution success Transaction successful
1A01	Good	EX	ACK received Transaction successful, ACK received
1A10	Good	EX	Data received Received host message on Read operation
4200	Warning	CM	Not all config items copied Not all configuration items copied
4201	Warning	CM	Rdr Cmd terminated config cmd Reader command terminated configuration command string
4202	Warning	CM	Client should not be notified
4502	Warning	RW	Input timeout
4503	Warning	RW	No input data
450E	Warning	RW	The app should be notified The application should be notified
4529	Warning	RW	Input from source other than requested Input from source is other than the ones requested
452A	Warning	RW	RWTSR already loaded
452B	Warning	RW	Prepare for reboot command received
452C	Warning	RW	Cancel reboot command received
4600	Warning	CU	Warning buff canceled Buffer canceled
4601	Warning	CU	Comm Timeout warning Communications timeout
4602	Warning	CU	Comm had no client to cancel Communications had no client to cancel
4603	Warning	CU	Comm Util no PH for config No protocol handler to configure

Status Codes and Error Messages Listed in Numerical Order (continued)

Hex Code	Severity	Subsystem	Reader Message and Description
4604	Warning	CU	Prot handler not loaded Protocol handler is not yet loaded
4605	Warning	CU	Prot handler already receiving Protocol handler is already receiving
4606	Warning	CU	PH not active for link request Protocol handler is not active for link request
4A00	Warning	TM	Timer not active Timer not currently active
5001	Warning	KB	Expanded buffer installed Expanded keyboard buffer installed or disabled
5201	Warning	KP	Backdoor Comms failed Backdoor communications failure
5A02	Warning	EX	NAK received Transaction failed, NAK received
5A07	Warning	EX	No more log records Standby file is empty
5A11	Warning	EX	Received data truncated Data was truncated
5A12	Warning	EX	Fields overflow Too many fields
5A20	Warning	EX	Record logged Record successfully stored in standby file
5A21	Warning	EX	Logged but upload error Record stored in standby file, but upload standby error
8100	Error	RS	Rservice invalid request code Invalid function code received
8101	Error	RS	Rservice not installed Reader services is not installed
8102	Error	RS	Rservice resume failed Reader services resume failed
8103	Error	RS	Rservice or App out of date Reader services or application is out of date
8104	Error	RS	RWTSR not connected Reader Wedge TSR is not installed

Status Codes and Error Messages Listed in Numerical Order (continued)

Hex Code	Severity	Subsystem	Reader Message and Description
8200	Error	CM	Invalid Object
8201	Error	CM	Invalid subfunction
8202	Error	CM	Invalid Param length Invalid parameter length
8203	Error	CM	Invalid Param val Invalid parameter value
8204	Error	CM	Invalid Param 1 value Invalid first parameter value
8205	Error	CM	Invalid Code 39 config Invalid Code 39 configuration (HIBC and no check digit)
8206	Error	CM	Invalid param 2 value Invalid second parameter value
8207	Error	CM	Invalid Code 39 config Invalid Code 39 configuration (HIBC and full ASCII)
8208	Error	CM	Invalid Code 39 config Invalid Code 39 configuration (HIBC and mixed full ASCII)
8209	Error	CM	Invalid Param 3 value Invalid third parameter value
820A	Error	CM	Invalid Codabar cfg Invalid Codabar configuration (ABC and discard start/stop)
820B	Error	CM	Protocol not loaded
820C	Error	CM	Invalid comm or protcl Invalid port or protocol specified
820D	Error	CM	No end quote No ending quote found
820E	Error	CM	No dbl quote No double quote (might be missing an opening quote)
820F	Error	CM	Invalid parser state
8210	Error	CM	Invalid config cmd Invalid configuration command
8211	Error	CM	Client returned err Client deemed configuration command invalid
8212	Error	CM	Invalid beep duration

Status Codes and Error Messages Listed in Numerical Order (continued)

Hex Code	Severity	Subsystem	Reader Message and Description
8213	Error	CM	Invalid beep freq Invalid beep frequency
8214	Error	CM	Invalid beep vol Invalid beep volume
8215	Error	CM	Beep vol alrdy off Beep volume already off
8216	Error	CM	Beep vol alrdy on Beep volume already on
8217	Error	CM	Invalid beep specifier Invalid beep specifier (must be “L” or “H”)
8218	Error	CM	Inappropriate protcl Inappropriate configuration for the current protocol
8219	Error	CM	BIOS rejected set config BIOS rejected the set configuration command
821A	Error	CM	Bld string too small Configuration command string object is too small
821B	Error	CM	Invalid Build cmd Configuration ID is invalid
821C	Error	CM	Invalid Build param Invalid build parameter
821D	Error	CM	No Build Func No build function
821E	Error	CM	No more build entries
821F	Error	CM	No more WM bld entries No more build entries in wedge lookup table
8220	Error	CM	No more WM bld entries No more build entries in wedge lookup table for “build next” WM command
8221	Error	CM	No LUM key entries No entries found in the LUM key table
8222	Error	CM	WM entry not found No match in lookup table for specified character
8223	Error	CM	Invalid Temp config At least one parameter rejected by client (see JANUS.ERR)

Status Codes and Error Messages Listed in Numerical Order (continued)

Hex Code	Severity	Subsystem	Reader Message and Description
8224	Error	CM	Config Item not found Configuration item is not found
8225	Error	CM	File Open err File open error
8226	Error	CM	File Write err File write error
8227	Error	CM	File Read err File read error
8228	Error	CM	Read record too large Record too large to read
8229	Error	CM	Invalid Hex Char in esc seq Invalid hex character found in the escape sequence
822A	Error	CM	Invalid esc sequence Invalid escape sequence
822B	Error	CM	Read file not opened Read attempted on a file that was not opened
822C	Error	CM	Write file not opened Write attempted on a file that was not opened
822D	Error	CM	File close op fail File close operation failed
822E	Error	CM	Attempt close non-open file Attempt made to close a file that was not opened
822F	Error	CM	Invalid default config Invalid default configuration
8230	Error	CM	RF back not installd RF back is not installed
8231	Error	CM	RF driver not installd RF protocol handler is not installed
8232	Error	CM	Path (IMPATH) not found Path specified by IMPATH is not found
8233	Error	CM	Bld BIOS val out of range BIOS returned an invalid value during build
8234	Error	CM	Config Item not allowed by Prot Configuration item is not allowed for the selected protocol

Status Codes and Error Messages Listed in Numerical Order (continued)

Hex Code	Severity	Subsystem	Reader Message and Description
8235	Error	CM	Config Item value out of range Configuration item value is out of range for the selected protocol
8236	Error	CM	Invalid 4th param value Invalid fourth parameter value
8237	Error	CM	Invalid 5th param value Invalid fifth parameter value
8238	Error	CM	Invalid 6th param value Invalid sixth parameter value
8239	Error	CM	Can't open JANUS.ERR Unable to open JANUS.ERR file
823A	Error	CM	Radio.cfg has invalid checksum File checksum is invalid
823B	Error	CM	Invalid RF freq in RF config Invalid RF frequency specified
823C	Error	CM	Invalid RF channel in RF config Invalid channel specified
823D	Error	CM	CM_ENCODED_RF_DATA_S != CM_NUM_RF_CHANNELS Structure mismatch
823E	Error	CM	Allowed RF channels after unallowed channel Specified RF channel is not allowed
823F	Error	CM	Ping cmd not allowed here Ping command not allowed here
8240	Error	CM	Ping cmd not xmitted, prot busy Ping command not transmitted (protocol busy)
8241	Error	CM	Missing end of config cmd Missing end of configuration command
8242	Error	CM	RF freq doesn't match RF back type RF frequency does not match the RF back
8243	Error	CM	Can't read RF back config data Unable to read configuration data from the RF back
8244	Error	CM	Unallowed video display mode Video display mode is not allowed
8245	Error	CM	Cmd source not allowed Command source is not allowed

Status Codes and Error Messages Listed in Numerical Order (continued)

Hex Code	Severity	Subsystem	Reader Message and Description
8246	Error	CM	Invalid 7th param value Invalid seventh parameter value
8300	Error	SC	Scanner load err Scanner load error
8301	Error	SC	Insufficient timers Not enough timers for the scanner to initialize
8302	Error	SC	Insufficient memory Insufficient memory for the scanner to initialize
8303	Error	SC	IPM connect failed Scanner not connected to power management
8304	Error	SC	Invalid Laser Timeout Invalid laser timeout configuration
8305	Error	SC	Counts obj integrity failed Count object integrity check failed
8306	Error	SC	Scanner vector table corrupt
8402	Error	DC	Autodiscrim table full Autodiscrimination table full
8403	Error	DC	Insufficient resources for decode
8404	Error	DC	Decode init failure Decode initialization failure due to scanner
8405	Error	DC	No scanner Decode initialization failure - no scanner present
8406	Error	DC	Invalid decodes cmd Invalid decodes command
8407	Error	DC	Symbology out of range Invalid symbology specified
840D	Error	DC	Can't decode scan Unable to decode scan
840E	Error	DC	Missing start/stop Missing start/stop character
840F	Error	DC	Too few counts to decode
8410	Error	DC	Invalid char Invalid character found
8411	Error	DC	Invalid acceleration

Status Codes and Error Messages Listed in Numerical Order (continued)

Hex Code	Severity	Subsystem	Reader Message and Description
8412	Error	DC	Insufficient chars for valid label Insufficient characters for valid label
8413	Error	DC	Invalid check digit
8414	Error	DC	Output string too short
8415	Error	DC	No leading margin
8416	Error	DC	Invalid start/stop
8417	Error	DC	Attempted decode past end of buffer Attempted decode past end of buffer (not enough counts for whole label)
8418	Error	DC	No trailing margin
8419	Error	DC	Invalid UPC supplemental
841A	Error	DC	Invalid parity
841B	Error	DC	Guard char missing Guard character missing
841C	Error	DC	Invalid row number
841D	Error	DC	Unable to scale counts
841E	Error	DC	Invalid 2 of 5 label
841F	Error	DC	Invalid 2 of 5 length
8420	Error	DC	2 of 5 label length exceeds maximum
8423	Error	DC	No valid label region found
8424	Error	DC	Ink spread exceeded threshold
8425	Error	DC	Denominator of expression zero Denominator of an expression is zero (divide by zero attempted)
8426	Error	DC	ASCII conversion failed Conversion to ASCII of full label failed
8501	Error	RW	Input request err Input request error
8504	Error	RW	Illegal RW mode Illegal reader commands parsed
8507	Error	RW	Rdr cmd parsing err Reader commands parsing error

Status Codes and Error Messages Listed in Numerical Order (continued)

Hex Code	Severity	Subsystem	Reader Message and Description
8508	Error	RW	Invalid config Invalid configuration
850F	Error	RW	Error in rdr cmd edit Error in reader command edit
8511	Error	RW	Unable to allocate memory for input
8512	Error	RW	Prot handler not linked Protocol handler not linked
8514	Error	RW	Rdr cmd parsing err on exit accum Reader command parsing error on exit accumulation
8515	Error	RW	Application break detected
8516	Error	RW	Invalid Rdr Wedge request Invalid Reader Wedge request
851B	Error	RW	Rdr cmd erroneously parsed A reader command was incorrectly parsed when Enter was parsed
851F	Error	RW	Label not accepted: App not requesting input Label not accepted because the scan-ahead is not enabled and the application is not at an input statement.
8520	Error	RW	Link failed, no buffers Reader Wedge link failed
8522	Error	RW	Invalid port value for input request Invalid port specified in input request
8523	Error	RW	Xmit buffer busy or Comm Utility err returned Communications error
8524	Error	RW	Exit Accum with Comms Error Communications error from an exit accumulation command
8526	Error	RW	Invalid option in RW_DO Invalid port specified in input request
8527	Error	RW	No label symbology, no label input yet No symbology code available
8528	Error	RW	Bad parameter
8600	Error	CU	Invalid config Invalid configuration
8601	Error	CU	Buffer length 0 error
8602	Error	CU	Comm port in use

Status Codes and Error Messages Listed in Numerical Order (continued)

Hex Code	Severity	Subsystem	Reader Message and Description
8603	Error	CU	Protocol error
8604	Error	CU	Comm port error
8605	Error	CU	Comm port busy
8606	Error	CU	Service not supported Communications request is not supported
8607	Error	CU	Prot handler already loaded Protocol handler already loaded
8608	Error	CU	Prot buffer error Protocol buffer error
8609	Error	CU	Unknown service request
860A	Error	CU	No data available
860B	Error	CU	ComUtil not loaded Communications utility is not loaded
860C	Error	CU	Resume/Suspend failure
860D	Error	CU	INT 14 already in use Communications utility INT 14 is already in use
860E	Error	CU	Incompatible rev PH or ComUtil Incompatible revision between the protocol handler and the communications utility
860F	Error	CU	Invalid EOF char
8610	Error	CU	Buffer must be > 256 bytes
8611	Error	CU	Prot handler not active Protocol handler is not active
8612	Error	CU	Buff size too big
8613	Error	CU	Re-entrant request for service
8614	Error	CU	H/W I/O error Hardware I/O error
86F0	Error	CU	Non-supported Comm service Non-supported communications service
8821	Error	PM	Cannot change state
8822	Error	PM	State mgr invalid cmd Invalid state manager command

Status Codes and Error Messages Listed in Numerical Order (continued)

Hex Code	Severity	Subsystem	Reader Message and Description
8842	Error	PM	IPM interface invalid cmd Invalid IPM interface command
8843	Error	PM	Parameter out of range
8844	Error	PM	Semaphore max value exceeded Semaphore maximum value exceeded
8A00	Error	TM	Invalid timer func cod Invalid timer function code
8A01	Error	TM	No free timers available
8A02	Error	TM	Illegal timer mode specified
8A03	Error	TM	Timer not allocated
8A04	Error	TM	Timer currently active
8A05	Error	TM	Invalid timer delay specified
8A06	Error	TM	Invalid timer id specified
8A07	Error	TM	Timer Resume/Suspend failure
8A08	Error	TM	Cannot process user request Timer utility cannot safely process user request
8B00	Error	BP	Pitch out of range
8B01	Error	BP	Duration out of range
8B02	Error	BP	Vol out of range Volume out of range
8B03	Error	BP	Not seq or preem beep Not a sequential or preemptive beep
8B04	Error	BP	Invalid battery flag
8B05	Error	BP	No preem timer available No preemptive timer available
8B06	Error	BP	No seq timer available No sequential timer available
8B07	Error	BP	Sequence empty
8B08	Error	BP	Beep sequence not started
8B09	Error	BP	Error init beep list Error initializing beep list

Status Codes and Error Messages Listed in Numerical Order (continued)

Hex Code	Severity	Subsystem	Reader Message and Description
8B0A	Error	BP	Beep avail list empty Beep available list empty
8B0B	Error	BP	Bad beep mgr reques Bad beep manager request
8B0C	Error	BP	Beep sequence full
8B0D	Error	BP	Beep in suspend state Beeper in suspend state
8B0E	Error	BP	Beeper Resume/Suspend failure
8E00	Error	IM	Substring not found
8E01	Error	IM	Extract string empty
8E02	Error	IM	Extract string too long
8E03	Error	IM	Append string overflow
8E04	Error	IM	Search string not found
8E05	Error	IM	Search string empty
8E06	Error	IM	Search string too long
8E07	Error	IM	String copy too long
8E08	Error	IM	Invalid parameter
8E09	Error	IM	Hex conversion length error
8E0A	Error	IM	Hex conversion overflow
8E0B	Error	IM	Invalid binary to ASCII radix
8E0C	Error	IM	Binary to ASCII field too small
8F00	Error	LG	Event queue empty
8F01	Error	LG	Prohibited area disable
8F02	Error	LG	Reader service disable
8F03	Error	LG	Configuration management disable
8F04	Error	LG	Scanner disable
8F05	Error	LG	Decodes disable
8F06	Error	LG	Reader Wedge disable
8F07	Error	LG	Communications utility disable
8F08	Error	LG	Protocol handler disable
8F09	Error	LG	Power management disable

Status Codes and Error Messages Listed in Numerical Order (continued)

Hex Code	Severity	Subsystem	Reader Message and Description
8F0A	Error	LG	BIOS disable
8F0B	Error	LG	Timer disable
8F0C	Error	LG	Beeper disable
8F0D	Error	LG	IRL desktop disable
8F0E	Error	LG	Prompting configuration disable
8F0F	Error	LG	Intermec library disabled
8F10	Error	LG	Event logger disable
8F11	Error	LG	Keyboard disable
8F12	Error	LG	System configuration disable
8F13	Error	LG	Keypad disable
8F14	Error	LG	Display disable
8F15	Error	LG	Communications manager disable
8F16	Error	LG	Workbench disable
8F17	Error	LG	Severity success disable
8F18	Error	LG	Severity warning disable
8F19	Error	LG	Severity error disable
8F1A	Error	LG	Severity fatal disable
8F1B	Error	LG	Wrong subsystem ID entered
8F1C	Error	LG	Wrong severity ID entered
9002	Error	KB	Can't disable not empty buffer Cannot disable because buffer is not empty
9003	Error	KB	Keypad buffer full Keyboard buffer is full, entire string rejected
9004	Error	KB	Expanded buffer not installed Expanded keyboard buffer not installed
9005	Error	KB	Invalid mode value was passed
9006	Error	KB	Exp buffer flush not install Expanded keyboard buffer flush failed
9007	Error	KB	Undefined message
9008	Error	KB	16.B3 err:Source str > dest bufr Source string exceeds source or destination buffer size

Status Codes and Error Messages Listed in Numerical Order (continued)

Hex Code	Severity	Subsystem	Reader Message and Description
9009	Error	KB	16.B3 err:Unknown str descriptor String descriptor type is unknown
900C	Error	KB	Invalid option requested from int 16 B4xxh
9100	Error	SS	Invalid service number
9101	Error	SS	ASIC bit number out of range
9102	Error	SS	Unknown ASIC I/O address
9103	Error	SS	Undefined message code
9104	Error	SS	Invalid COM Port id Invalid communications port ID
9105	Error	SS	Invalid Args Invalid Argument
9200	Error	KP	Timeout mid. transaction Timeout occurred in the middle of a transaction
9201	Error	KP	Timeout 1st transaction Timeout occurred on the first transaction
9202	Error	KP	Keypad busy
9203	Error	KP	Forced transaction abort
9204	Error	KP	Failed to process command
9205	Error	KP	Invalid func number Invalid function number
9206	Error	KP	Invalid keypad comms status Invalid keypad communications status
9207	Error	KP	Passed in wrong parameter
9208	Error	KP	BIOS returned invalid value
9300	Error	DP	Abs value out range Absolute value out range
9301	Error	DP	Invalid display size mode
9302	Error	DP	Display function undefined
9303	Error	DP	Invalid backlight timeout value
9304	Error	DP	Display BIOS returned an invalid status
9305	Error	DP	Invalid viewport mode
9306	Error	DP	Invalid scroll mode

Status Codes and Error Messages Listed in Numerical Order (continued)

Hex Code	Severity	Subsystem	Reader Message and Description
9307	Error	DP	Invalid video mode
9308	Error	DP	Invalid size mode
9309	Error	DP	Invalid height mode Invalid character height mode
9A03	Error	EX	Transmit fail
9A04	Error	EX	Receive timeout
9A05	Error	EX	Packet header error
9A06	Error	EX	Transaction ID missing
9A08	Error	EX	Standby file missing
9A09	Error	EX	Standby file I/O error
9A40	Error	EX	Buffer overflow Host response buffer is too small
9A41	Error	EX	Buffer overflow and upload standby error

Status Codes Listed by Subsystem

The status codes in the following tables are organized by subsystems. To find a particular status code, identify the subsystem by looking at bits 8 through 13 and refer to that section:

Code	Abbrev.	Name
0100	RS	Reader Services
0200	CM	Configuration Management
0300	SC	Scanner
0400	DC	Decodes
0500	RW	Reader Wedge
0600	CU	Communications
0800	PM	Power Management
0A00	TM	Timer
0B00	BP	Beep
0E00	IM	Intermec Library
0F00	LG	Event Logger
1000	KB	Keyboard Buffer
1100	SS	System Configuration
1200	KP	Keypad Services
1300	DP	Display
1A00	EX	DCM Support

Subsystem 0100 RS (Reader Services)

Hex Code	Message
0100	Reader service successful
0103	Reader services resume successful
8100	Invalid function Hex code received
8101	Reader services is not installed
8102	Reader services resume failed
8103	Reader services or application is out of date
8104	Reader Wedge TSR is not installed

Subsystem 0200 CM (Configuration Management)

Hex Code	Message
0200	Request successful
0201	End of record encountered
0202	End of file encountered
4200	Not all configuration items copied
4201	Reader command terminated configuration command string
4292	Client should not be notified
8200	Invalid object
8201	Invalid subfunction
8202	Invalid parameter length
8203	Invalid parameter value
8204	Invalid first parameter
8205	Invalid Code 39 configuration (HIBC and no check digit)
8206	Invalid second parameter
8207	Invalid Code 39 configuration (HIBC and full ASCII)
8208	Invalid Code 39 configuration (HIBC and mixed full ASCII)
8209	Invalid third parameter
820A	Invalid Codabar configuration (ABC and discard start/stop)
820B	Protocol not loaded
820C	Invalid port or protocol specified
820D	No ending quote found
820E	No double quote (might be missing an opening quote)
820F	Invalid parser state
8210	Invalid configuration command
8211	Client deemed configuration command invalid
8212	Invalid beep duration
8213	Invalid beep frequency
8214	Invalid beep volume
8215	Beep volume already off
8216	Beep volume already extra loud
8217	Invalid beep specifier (must be “L” or “H”)

Subsystem 0200 CM (Configuration Management) (continued)

Hex Code	Message
8218	Inappropriate configuration for the current protocol
8219	BIOS rejected the set configuration command
821A	Configuration command string object is too small
821B	Configuration ID is invalid
821C	Invalid build parameter
821D	No build function
821E	No more build entries
821F	No more build entries in wedge lookup table
8220	No more build entries in wedge lookup table for “build next” WM command
8221	No entries found in the LUM key table
8222	No match in lookup table for specified character
8223	At least one parameter rejected by client (see JANUS.ERR)
8224	Configuration item is not found
8225	File open error
8226	File write error
8227	File read error
8228	Record too large to be read
8229	Invalid hex character found in the escape sequence
822A	Invalid escape sequence
822B	Read attempted on a file that was not opened
822C	Write attempted on a file that was not opened
822D	File close operation failed
822E	Attempt made to close a file that is not open
822F	Invalid default configuration
8230	RF back is not installed
8231	RF protocol handler is not installed
8232	Path specified by IMPATH is not found
8233	BIOS returned an invalid value during build
8234	Configuration item is not allowed for the selected protocol

Subsystem 0200 CM (Configuration Management) (continued)

Hex Code	Message
8235	Configuration item value is out of range for the selected protocol
8236	Invalid fourth parameter value
8237	Invalid fifth parameter value
8238	Invalid sixth parameter value
8239	Unable to open JANUS.ERR
823A	File checksum invalid
823B	Invalid RF frequency
823C	Invalid channel is specified
823D	Structure mismatch
823E	Specified RF channel is not allowed
823F	Ping command not allowed here
8240	Ping command not transmitted (protocol busy)
8241	Missing end of configuration command
8242	RF frequency does not match the RF back type
8243	Unable to read configuration data from the RF back
8244	Video display mode is not allowed
8245	Command source is not allowed

Subsystem 0300 SC (Scanner)

Hex Code	Message
0300	Scan successful
8300	Scanner load error
8301	Not enough timers for the scanner to initialize
8302	Insufficient memory for the scanner to initialize
8303	Scanner cannot connect to power management
8304	Invalid laser timeout configuration
8305	Counts object integrity check failed
8306	Scanner vector table is corrupt

Subsystem 0400 DC (Decodes)

Hex Code	Message
0400	Successful command
0401	Symbology is already enabled
0402	Symbology is not enabled
040A	Successful decode
040B	Intermediate row reread
040C	Intermediate row decoded
0421	Default command symbology
0422	Mixed ASCII format
8402	Autodiscrimination table full
8403	Insufficient resources for decode
8404	Decode initialization failure due to scanner
8405	Decode initialization failure—no scanner
8406	Invalid decodes command
8407	Invalid symbology specified
840D	Unable to decode scan
840E	Missing start/stop character
840F	Too few counts to decode
8410	Invalid character found
8411	Invalid acceleration
8412	Insufficient characters for valid label
8413	Invalid check digit
8414	Output string too short
8415	No leading margin
8416	Invalid start/stop
8417	Attempted decode past end of buffer (not enough counts for the whole label)
8418	No trailing margin
8419	Invalid UPC supplemental
841A	Invalid parity
841B	Guard character missing

Subsystem 0400 DC (Decodes) (continued)

Hex Code	Message
841C	Invalid row number
841D	Unable to scale counts
841E	Invalid 2 of 5 label
841F	Invalid 2 of 5 length
8420	2 of 5 label length exceeds maximum
8423	No valid label region found
8424	Ink spread exceeded threshold
8425	Denominator of an expression is zero
8426	Conversion to ASCII of full label failed

Subsystem 0500 RW (Reader Wedge)

Hex Code	Message
0500	Reader Wedge successful
0505	No reader commands parsed
0506	Valid reader commands parsed
0509	Accumulating reader command
050A	Reader command override
050B	Reader command enter accumulation
050C	Reader command exit accumulation
050D	Accumulating multiple-read labels
0510	An Enter reader command was parsed
0513	Protected field has been parsed
0517	Keycode label has been parsed
0518	An isolated Enter command was parsed
0519	A successful reader command parse on exit from accumulation
051A	A reader command has been successfully parsed when Enter is parsed
051C	An Enter Accumulate has been parsed when accumulating
051D	A label terminating a multiple-read sequence has been scanned
051E	A label beginning a multiple-read sequence has been scanned
0521	No communications status to report
0525	Reader commands forwarded to the application have been parsed
4502	Input timeout
4503	No input data
450E	The application should be notified
4529	Input from the source is other than the ones requested
452A	RWTSR is already loaded
452B	Prepare for reboot command received
452C	Cancel reboot command received
8501	Input request error
8504	Illegal reader commands parsed
8507	Reader commands parsing error
8508	Invalid configuration error

Subsystem 0500 RW (Reader Wedge) (continued)

Hex Code	Message
850F	Error in reader command edit
8511	Unable to allocate memory for input
8512	Protocol handler is not linked
8514	Reader command parse error on exit accumulation
8515	Application break detected
8516	Invalid Reader Wedge request
851B	A reader command has been erroneously parsed when Enter is parsed
851F	Label not accepted because the scan-ahead is not enabled and application is not at an input statement
8520	Reader Wedge link failed
8522	Invalid port specified in input request
8523	Communications error
8524	Communications error from an exit accumulation command
8526	Library coding error
8527	No symbology code available
8528	Bad parameter

Subsystem 0600 CU (Communications)

Hex Code	Message
0600	Communications operation success
0601	Communications buffer done
4600	Buffer canceled
4601	Communications timeout
4602	Communications had no client to cancel
4603	No protocol handler to configure
4604	Protocol handler is not yet loaded
4605	Protocol handler already receiving
4606	Protocol handler not active for link request
8600	Invalid configuration
8601	Buffer length zero error
8602	Communications port is in use
8603	Protocol error
8604	Communications port error
8605	Communications port is busy
8606	Communications request is not supported
8607	Protocol handler already loaded
8608	Protocol buffer error
8609	Unknown service request
860A	No data available
860B	Communications utility is not loaded
860C	Resume/suspend failure
860D	Communications utility INT 14 is already in use
860E	Incompatible revision between the protocol handler and the communications utility
860F	Invalid end of file character
8610	Buffer must be 256 bytes or greater
8611	Protocol handler not active
8612	Buffer size is too big

Subsystem 0600 CU (Communications) (continued)

Hex Code	Message
8613	Reentrant request for service
8614	Hardware I/O error
86F0	Unsupported communications service

Subsystem 0800 PM (Power Management)

Hex Code	Message
0800	Power management success
8821	Cannot change state
8822	Invalid state manager command
8842	Invalid IPM interface command
8843	Parameter out of range
8844	Semaphore maximum exceeded

Subsystem 0A00 TM (Timer)

Hex Code	Message
0A00	Timer operation success
4A00	Timer not currently active
8A00	Invalid timer function code
8A01	No free timers available
8A02	Illegal timer mode specified
8A03	Timer not allocated
8A04	Timer is currently active
8A05	Invalid timer delay specified
8A06	Invalid timer ID specified
8A07	Timer resume/suspend failure
8A08	Timer utility cannot safely process user request

Subsystem 0B00 BP (Beep)

Hex Code	Message
0B00	Beeper call success
8B00	Pitch out of range
8B01	Duration out of range
8B02	Volume out of range
8B03	Not a sequential or a preemptive beep
8B04	Invalid battery flag
8B05	No preemptive timer available
8B06	No sequential timer available
8B07	Sequence empty
8B08	Beep sequence not started
8B09	Error initializing beep list
8B0A	Beep available list empty
8B0B	Bad beep manager request
8B0C	Beep sequence full
8B0D	Beeper in suspend state
8B0E	Beeper resume/suspend failure

Subsystem 0E00 IM (Intermec Library)

Hex Code	Message
0E00	String extract complete
0E01	String append complete
0E02	String search found
0E03	String copy complete
0E04	Successful operation
8E00	Substring not found
8E01	String extract empty
8E02	String extract too long
8E03	String append overflow
8E04	String search not found
8E05	String search empty
8E06	String search too long
8E07	String copy too long
8E08	Invalid parameter
8E09	Hex conversion length error
8E0A	Hex conversion overflow
8E0B	Invalid binary to ASCII radix
8E0C	Binary to ASCII field too small

Subsystem 0F00 LG (Event Logger)

Hex Code	Message
0F00	Successful logger operation
8F00	Event queue empty
8F01	Prohibited area disable
8F02	Reader service disable
8F03	Configuration management disable
8F04	Scanner disable
8F05	Decodes disable
8F06	Reader Wedge disable
8F07	Communications utility disable
8F08	Protocol handler disable
8F09	Power management disable
8F0A	BIOS disable
8F0B	Timer disable
8F0C	Beeper disable
8F0D	IRL desktop disable
8F0E	Prompting configuration disable
8F0F	Intermec library disabled
8F10	Event logger disable
8F11	Keyboard disable
8F12	System configuration disable
8F13	Keypad disable
8F14	Display disable
8F15	Communications manager disable
8F16	Workbench disable
8F17	Severity success disable
8F18	Severity warning disable
8F19	Severity error disable
8F1A	Severity fatal disable
8F1B	Wrong subsystem ID entered
8F1C	Wrong severity ID entered

Subsystem 1000 KB (Keyboard Buffer)

Hex Code	Message
1000	Expanded keyboard buffer success
100A	Expanded buffer enabled
100B	Expanded buffer disabled
5001	Expanded keyboard buffer installed or disabled
9002	Cannot disable because buffer is not empty
9003	Keyboard buffer is full (entire string rejected)
9004	Expanded keyboard buffer not installed
9005	Invalid mode value was passed
9006	Expanded keyboard buffer flush not successful
9007	Undefined message
9008	Source string exceeds source or destination buffer size
9009	String descriptor type is unknown
900C	Invalid option requested from INT 16 B4xxh

Subsystem 1100 SS (System Configuration)

Hex Code	Message
1100	System configuration successful
9100	Invalid service number
9101	ASIC bit number is out of range
9102	Unknown ASIC I/O address
9103	Undefined message code
9104	Invalid communications port ID
9105	Invalid argument

Subsystem 1200 KP (Keypad Services)

Hex Code	Message
1200	Successful keypad operation
5201	Backdoor communications failure
9200	Timeout occurred in the middle of a transaction
9201	Timeout occurred on the first transaction
9202	Keypad busy
9203	Forced transaction abort
9204	Failed to process a command
9205	Invalid function number
9206	Invalid keypad communications status
9207	Wrong parameter passed in
9208	BIOS returned an invalid value

Subsystem 1300 DP (Display)

Hex Code	Message
1300	Successful display operation
9300	Absolute value out of range
9301	Invalid display size mode
9302	Display function undefined
9303	Invalid backlight timeout value
9304	Display BIOS returned an invalid status
9305	Invalid viewport mode
9306	Invalid scroll mode
9307	Invalid video mode
9308	Invalid size mode
9309	Invalid character height mode

Subsystem 1A00 EX (DCM Support)

Hex Code	Message
1A00	Execution success
1A01	ACK received
1A10	Data received
5A02	NAK received
5A07	No more log records
5A11	Received data truncated
5A12	Fields overflow
5A20	Record logged
5A21	Logged but upload error
9A03	Transmit fails
9A04	Receive timeout
9A05	Packet header error
9A06	Transaction ID is missing
9A08	Standby file is missing
9A09	Standby file I/O error
9A40	Buffer overflow
9A41	Buffer overflow and upload error

B

Sample Interrupt Programs

This appendix lists C language programs that demonstrate how to use interrupts.

Using the Sample Programs

The sample programs in this appendix demonstrate how to use software interrupts in place of the Intermec PSK functions. It is much easier to write programs that use the PSK functions than to write programs that directly call the software interrupts. For more information on using software interrupts, see Chapter 3, "Software Interrupts."

Note: *Do* ***not*** *run programs that use* ***PSK library functions*** *on your PC. If you attempt to run these programs, you will receive an error message and the program will not run.*

Caution
Do not run programs that use **Intermec-specific interrupt extensions** ***on your PC. If you attempt to run these programs, they will cause your PC to lock up and possibly corrupt your system BIOS.***

Conseil
N'exécutez pas de programmes utilisant **des extensions d'interruption spécifiques à Intermec** ***sur votre PC. Si vous tentez de les exécuter sans le Simulator, ces programmes risquent de verrouiller votre PC et d'altérer le BIOS de votre système.***

apmif.c

```
/* apmif.c
 *
 * This program is an APM interface test module. It allows entry of the APM
 * interface registers from the MS-DOS command line, and then it makes an APM call.
 * The results displayed are the register contents returned from the call.
 *
 * Compiler: Borland C++ 3.11
 */

typedef unsigned char uchar;
typedef unsigned int    uint;

#include <stdio.h>
#include <ctype.h>
#include <string.h>

uint tohex(char*);
void get_data(char**);
void call_APM(void);
void print_IV(void);

uint AX_REG,BX_REG,CX_REG,DX_REG,CARRY;

void main(int argc,char **argv){

    AX_REG = 0x5300;                //check for pm enabled
    BX_REG = 0x0;
    CX_REG = 0x0;
    DX_REG = 0x0;
    call_APM();

    if (BX_REG != 0x504D){          //504D = PM, means apm implemented
     printf("APM not supported.\n\n");
     return;
    }

    if ((CX_REG & 0x0008) != 0)     //check for pm disabled
        printf("PM is disabled.\n\n");

    AX_REG = 0x5301;                // connect as apm client
    BX_REG = 0x0;
    CX_REG = 0x0;
    DX_REG = 0x0;
    call_APM();

    if (CARRY != 0){
        printf("APM client already connected.\n\n");
    }
    else{
        printf("Successful connect.\n\n");
    }


    if (argc == 1)                  // if no arguments, prompt for them
        printf("Enter AX BX CX...\n\n");

    get_data(argv);
```

```
                                    //display register values entered

    printf("i AX=%0.4X BX=%0.4X\ni CX=%0.4X DX=%0.4X\n",
                    AX_REG,BX_REG,CX_REG,DX_REG);

    call_APM();                     //do the apm call

/* have to call standby twice to make it work (so it won't skip activity)*/

    if ((AX_REG == 0x5307)&&(CX_REG == 0x0001))
        call_APM();

                                    //display the register values returned

    printf("o AX=%0.4X BX=%0.4X\no CX=%0.4X DX=%0.4X\n",
        AX_REG,BX_REG,CX_REG,DX_REG);

    printf("carry=%0.4X\n\n",CARRY);

    if (AX_REG == 0x530B){          //if get pm event and standby, do it
        if ((BX_REG & 0x01) != 0x0){
            AX_REG = 0x5307;
            BX_REG = 0x0001;
            CX_REG = 0x0001;
            call_APM();
            call_APM();
        }
    }
}

void call_APM(){    /* the call apm interface routine */
asm{
    push ax
    push bx
    push cx
    push dx
    mov ax,AX_REG
    mov bx,BX_REG
    mov cx,CX_REG
    mov dx,DX_REG
    int 15h
    mov AX_REG,ax
    mov BX_REG,bx
    mov CX_REG,cx
    mov DX_REG,dx
    pop dx
    pop cx
    pop bx
    pop ax
   } /* end asm */
    asm mov CARRY,0
    asm jnc a
    asm inc CARRY
    a:
}

void get_data (char **dos_arg){   /* get the register values from user entry */
    char *regval;
    char charbuff[10];

    regval = *dos_arg++;
    if (*dos_arg == 0) {
```

```
        printf("Enter AX value in hex:");
        fflush(stdin);
        regval=charbuff;
        gets (regval);
    }
    else
    regval = *dos_arg++;

    AX_REG = tohex(regval);

    if (*dos_arg == 0) {
        printf("Enter BX value in hex:");
        fflush(stdin);
        regval=charbuff;
        gets (regval);
    }
    else
    regval = *dos_arg++;

    BX_REG = tohex(regval);

    if (*dos_arg == 0) {
        printf("Enter CX value in hex:");
        fflush(stdin);
        regval=charbuff;
        gets (regval);
    }
    else
    regval = *dos_arg++;

    CX_REG = tohex(regval);

    if (*dos_arg == 0) {
        DX_REG = 0;
    }
    else {
        regval = *dos_arg++;
        DX_REG = tohex(regval);
    }
}

uint tohex(char *string){   /* convert ascii characters to hex */
    uint    ctr;
    uint    num;
    uint    value = 0;
    uint    stop;

    stop=strlen(string);
    for (ctr = 0; ctr < stop; ctr++){
      value <<= 4;
      num = string[ctr];
      num -= '0';
      num = (num > 9) ? (num - ('A' - ':')) : num;
      num = (num > 15) ? (num - ('a' - 'A')) : num;
      value += num;
    }   /* end for */

    return value;
}
```

appbreak.cpp

```
/* appbreak.cpp
 *
 * This program demonstrates application break control using interrupts.
 *
 * The operator presses application break to exit program.  The application break sequence
 * is as follows:
 *
 *    Start the Appbreak program:
 *    1/0                turn off Janus
 *    F3+2+left arrow    at the same time
 *    1                  by itself
 *    1/0                turn Janus back on
 *
 * Written: Aug. 30, 1993
 * Last Modified: Aug. 30, 1993
 * Author: Ray Rogers
 * Compiler: Borland C++ 3.1
 */

#include  "dos.h"
#include  "stdio.h"
#include  "stdlib.h"
#include  "conio.h"
#include  "im20lib.h"

#define  KEY_Int        0x7e           // keyboard interrupt
#define  BREAK_REQ      0xff           // break requested
#define  TRANS_OK       0x00           // sucessful KSCPU transaction
#define  VALID_SERVICE  0x00           // valid service call to 0x7e
#define  ESC            0x1b           // escape key
#define  FOREVER        1              // endless loop control

typedef struct {
    IM_UCHAR   status;                 // KSCPU comm status
    IM_UCHAR   request;                // request made or not?
    } RESP_BUFF;

RESP_BUFF  rbuff;                      // KSCPU response buffer

// Prototypes
IM_UINT  Chk_App_Break(RESP_BUFF *);

void  main()
{
    clrscr();
    printf("%s", "\n\rDo appl. break or\n\rpress ESC\n\r");

    while(FOREVER)
        {
        if(VALID_SERVICE == Chk_App_Break(&rbuff))
            {
            if(rbuff.status == TRANS_OK)
                {
                if(rbuff.request == BREAK_REQ)
                    {
                    printf("%s", "\n\rAppl. Break detected!\n\r");
                    exit(0);
                    }
                }
```

```
        }                               // end VALID_SERVICE
        if(kbhit())
            if(getch() == ESC)          // just in case
                {
                printf("%s", "\n\rEscape was pressed!\n\r");
                exit(0);
                }
        }
}

// Check for application break request
IM_UINT  Chk_App_Break(RESP_BUFF *rbuff)
{
    _ES = FP_SEG((RESP_BUFF *) rbuff);   // far pointer to response buffer
    _BX = FP_OFF((RESP_BUFF *) rbuff);   // far pointer to response buffer

    _AH = 0x01;                          // check app break function

    geninterrupt(KEY_Int);

    return(_AH);
}
```

backlite.cpp

```
/* backlite.cpp
 *
 * This program demonstrates how to toggle the back light.
 *
 * Written: Aug. 12, 1993
 * Last Modified: Aug. 19, 1993
 * Author: Ray Rogers
 * Compiler: Borland C++ 3.1
 */

#include  "dos.h"

#define  LCD_Int    0x79                // LCD size mode manager interrupt

void  main()
{
    _AH = 0x01;                         // back light function
    _AL = 0x02;                         // toggle back light state

    geninterrupt(LCD_Int);
}
```

close.cpp

```
/* close.cpp
 *
 * This program demonstrates how to close COM2 on the 2D JANUS.
 *
 * Written: Feb. 10, 1999
 * Last Modified: Feb. 10, 1999
 * Author: Peter J. Girodat
 * Compiler: Borland C++ 3.1
 */

#include <dos.h>
#include <stdio.h>
#include <stdlib.h>

#define RS_VECTOR          0x7D      /* Interrupt 7DH Reader Services      */
#define RS_ASIC_RESET      0x0C      /* Function  0CH ASIC Reset           */
#define RS_CLOSE_COM2      0x0D      /* Function  0DH Close COM2           */
#define RS_ALL_SUCESS      0x0100    /* Reader Service sucessful           */
#define RS_INVALID_REQUEST 0x8100    /* Invalid function Hex code received */

void main()
{
   _AH = RS_CLOSE_COM2;             // AH = Function 0DH
   _AL = 0;

   geninterrupt(RS_VECTOR);

   if (_AX == RS_ALL_SUCESS)
      printf ("COM2 CLOSED\n");
   else if (_AX == RS_INVALID_REQUEST)
      printf ("JANUS 1D Reader\n");
}
```

contrast.cpp

```
/* contrast.cpp
 *
 * This program demonstrates contrast control using interrupts.  It increases the
 * contrast by one step each time the space bar is pressed.  It accepts a parameter
 * in the range of 0 - 31 from the command line and sets the contrast to that level.
 *
 * Written: Aug. 24, 1993
 * Last Modified: Aug. 24, 1993
 * Author: Ray Rogers
 * Compiler: Borland C++ 3.1
 */

#include  "stdio.h"
#include  "string.h"
#include  "dos.h"
#include  "conio.h"
#include  "stdlib.h"
#include  "im20lib.h"

// Defines
#define  FOREVER  1
#define  LCD_Int  0x79                  // LCD interrupt
#define  LCD_CONTRAST  0                // contrast function
#define  ESC  0x1b
#define  ZERO  '0'
#define  PERIOD  '.'

// Prototypes
IM_UCHAR  Step_Contrast(IM_UCHAR);
IM_UCHAR  Set_Contrast(IM_UCHAR);
IM_UCHA  Get_Contrast(void);

void  main(int  argc, char *argv[])
{
IM_UCHAR    ch, level;

    if(argc == 1)
        {
        clrscr();                       // no command line parameters
        printf("%s", "\n\rPress 0 to decrease\n\rperiod to increase\n\rESC to quit");
        }
    else
        {
        Set_Contrast(atoi(argv[1]));
        exit(0);
        }

    // main processing loop, exit loop on ESC key
    while(FOREVER)
        {
        if(kbhit())
            {
            ch = getch();
            switch(ch)
                {
                case  ESC:
                      clrscr();
                      exit(0);
```

```
                case  ZERO:
                      Step_Contrast(--(level = Get_Contrast()));
                      break;
                case  PERIOD:
                      Step_Contrast(++(level = Get_Contrast()));
                      break;
                };
            }
        }    // end main loop
}

// Get the current contrast level
IM_UCHAR  Get_Contrast()
{
    _AH = LCD_CONTRAST;
    _AL = 3;                            // current contrast

    geninterrupt(LCD_Int);

    return(_AL);
}

// Step the contrast one level by call with 0 parameter or set by
// supplying a value
IM_UCHAR  Step_Contrast(IM_UCHAR  step)
{
static  IM_UCHAR    once, level;

    if(!once)
        {
        once = 1;                       // allow Get_Contrast() only once
        level = Get_Contrast();
        }

        if(step != 0) level = step;     // use step value if provided
        if(level >= 31) level = 0;      // keep level within bounds (not below zero -
                                        // unsigned)
        if(step == 0) level++;

        return(Set_Contrast(level));
}

// Set the contrast to the requested level
IM_UCHAR  Set_Contrast(IM_UCHAR level)
{
    _AH = LCD_CONTRAST;
    _AL = 1;                            // set contrast to value function
    _BL = level;

    geninterrupt(LCD_Int);

    return(_AL);
}
```

ctlkey.cpp

```
/* ctlkey.cpp
 *
 * This program demonstrates Ctrl key control by using interrupts to enable/disable
 * the Ctrl key.
 *
 * Written: Sept. 1, 1993
 * Last Modified: Sept. 1, 1993
 * Author: Ray Rogers
 * Compiler: Borland C++ 3.1
 */

#include  "dos.h"
#include "stdlib.h"
#include  "conio.h"
#include  "stdio.h"
#include  "im20lib.h"

#define  KEYPAD_Int  0x7e                // key pad interrupt
#define  KP_CTL  0x08                    // keypad ctl key function
#define  KP_OFF  0x00                    // function is off
#define  KP_ON  0x01                     // function is on
#define  KP_STATUS  0x02                 // get status
#define  KP_ENABLED  0xff                // function is enabled
#define  VALID_SERVICE  0x00             // valid function call
#define  TRANS_OK  0x00                  // func transaction executed ok

typedef  struct  {
        IM_UCHAR  status;               // KSCPU comm status
        IM_UCHAR  kp_status;            // 0xff if function already enabled
        IM_UINT  padding;               // work space
        }  RESP_BUFF;

RESP_BUFF  rbuff;

// Prototypes
IM_UINT  KP_Func(IM_UCHAR, IM_UCHAR, RESP_BUFF *);

void  main()
{
    clrscr();

    if(VALID_SERVICE == KP_Func(KP_CTL, KP_STATUS, &rbuff)) // get current status of click
        if(TRANS_OK == rbuff.status)
            if(rbuff.kp_status == KP_ENABLED)              // if it is on...
                {
                KP_Func(KP_CTL, KP_OFF, &rbuff);           // turn it off
                printf("%s", "\n\rControl Key DISABLED\n\r\n\rPress a key...");
                }
            else
                {
                KP_Func(KP_CTL, KP_ON, &rbuff);  // else, turn it on
                printf("%s", "\n\rControl key ENABLED\n\r\n\rPress a key...");
                }

    while(!kbhit());                    // wait for a keypress
    getch();
    clrscr();
    exit(0);
}
```

ctlkey.cpp

```
// Execute requested keypad function
IM_UINT  KP_Func(IM_UCHAR func, IM_UCHAR subfunc, RESP_BUFF *resp)
{
    _ES = FP_SEG(resp);
    _BX = FP_OFF(resp);

    _AH = func;                          // keypad function
    _AL = subfunc;                       // subfunction

    geninterrupt(KEYPAD_Int);

    return(_AH);
}
```

exstat.cpp

```
/* exstat.cpp
 *
 * This program demonstrates JANUS extended protocol status reporting using interrupts.
 * This program also demonstrates program control of the JANUS viewport using interrupts.
 *
 * NOTE: For units with Radio Frequency (RF), this program is known to function properly
 * only with Version 1.24 firmware.
 * Written: July 25, 1994
 *
 * Last Modified: July 25, 1994
 * Author: Ray Rogers
 * Compiler: Borland C++ 3.1
 */

#include  "stdio.h"
#include  "string.h"
#include  "dos.h"
#include  "conio.h"
#include  "stdlib.h"
#include  "im20lib.h"

// Defines
#define  COM_Int  0x14                  // Communications interrupt
#define  LCD_Int  0x79                  // LCD size mode manager interrupt
#define  ESC  0x1b                      // char 27,  Escape
#define  SPACE  ' '                     // blank space
#define  ZERO  '0'                      // zero == set viewport to origin 0,0
#define  TRUE  1
#define  FOREVER  TRUE
#define  VP_ACROSS  5                   // number of viewport steps across
#define  VP_DOWN  4                     // number of viewport steps down
#define  VP_STEPSIZE  15                // columns to move viewport right
#define  OK  0x600                      // ok status
#define  ERR_NO_HANDLER 0x8611          // no protocol handler loaded
#define  ERR_BAD_VER  0x860e            // old firmware - requires 1.24 or later

// Global Variables
IM_UINT  vm_mode;                       // saves original video mode
IM_UINT  vm_params;                     // saves original video parameters
IM_UINT  vp = 0;                        // viewport position
IM_UINT  vp_vert = 1;                   // viewport vertical control
IM_COMM_STATUS_BUFFER_s  sb0, sb3;      // define buffers to hold status

// Prototypes
IM_UINT  Set_Video_Mode(void);
IM_UINT  Get_Ext_Status(IM_UINT, IM_COMM_STATUS_BUFFER_s *);
IM_UINT  VP_Move(IM_UINT, IM_UINT);
void  Display_Status(IM_UINT, IM_COMM_STATUS_BUFFER_s *, IM_UINT);
void  Move_Viewport(void);
void  err_msg(IM_UINT);

void  main()
{
IM_UCHAR  ch;
IM_UINT  status0, status3;

    Set_Video_Mode();

    sb0.status_structure_version = STATUS_STRUCT_VERSION;
```

```
    sb3.status_structure_version = STATUS_STRUCT_VERSION;
    status0 = Get_Ext_Status(0, &sb0);   // status for port 0
    status3 = Get_Ext_Status(3, &sb3);   // status for port 3

    clrscr();

    VP_Move(0, 0);

    printf("%s", "\n\rPress the space bar to\n\r");
    printf("%s", "move the viewport \n\ror ESC to quit");

    ch = 0x00;
    while(ch != SPACE && ch != ESC)
        if(kbhit()) ch = getch();

    clrscr();

    VP_Move(0, 0);

    Display_Status(0, &sb0, status0);
    printf("%s", "\r\n\n");
    Display_Status(3, &sb3, status3);

    VP_Move(0, 0);

    // Loop for moving the viewport to display the data
    while(FOREVER)
        {
        ch = getch();

        if(ch == SPACE)
            Move_Viewport();

        if(ch == ZERO)                  // just handy to have
            VP_Move(0, 0);

        if(ch == ESC)
            break;
        }

    clrscr();

    VP_Move(0, 0);
}

// Display a status buffer on the 80x25 screen
void  Display_Status(IM_UINT port, IM_COMM_STATUS_BUFFER_s *sb, IM_UINT status)
{
static  IM_UCHAR  *prot[] =
    {
    "User Defined",
    "Pt to Pt",
    "Polling Mode D",
    "Multi Drop",
    "Reserved1",
    "XModem",
    "Reserved2",
    "Dos Protocol",
    "Reserved3",
    "None",
    "No RF Protocol",
    "RF Protocol"
```

```
    };

if(sb->handler_type != IM_RF_BACK_PH && status == OK)  // if not the rf handler
    {
    printf("Port: %1d    Protocol Handler Ver. %s\n\r", port, sb->handler_ver);
    printf("Baud= %6lu   Parity= %1X  Stop Bits= %1X   Data Bits= %1X\n\r",
        sb->specific.serial.baud_rate, sb->specific.serial.parity,
        sb->specific.serial.stop_bits, sb->specific.serial.data_bits);
    printf("Modem Status= %2X  Port Status= %2X  Active Protocol= %s\n\r",
        sb->specific.serial.modem_status, sb->specific.serial.port_status,
        prot[sb->specific.serial.active_prot]);
    printf("LRC= %s  Interchar Delay= %2X  Turnaround Delay= %d\n\r",
        sb->specific.serial.lrc_enabled == 0 ? "No " : "Yes",
        sb->specific.serial.interchar_delay,
        sb->specific.serial.turnaround_delay);
    printf("POL char  = %3X   SEL char      = %2X     RES char   = %2X\n\r",
        sb->specific.serial.pol_char, sb->specific.serial.sel_char,
        sb->specific.serial.res_char);
    printf("REQ char  = %3X  AFF char  = %2X  NEG char  = %2X  SOM char= %2X\n\r",
        sb->specific.serial.req_char, sb->specific.serial.aff_char,
        sb->specific.serial.neg_char, sb->specific.serial.som_char);
    printf("TXEOM len = %3X   TXEOM char1   = %2X     TXEOM char2= %2X\n\r",
        sb->specific.serial.txeom_len, sb->specific.serial.txeom_char_1,
        sb->specific.serial.txeom_char_2);
    printf("RXEOM len = %3X   RXEOM char1   = %2X     RXEOM char2= %2X\n\r",
        sb->specific.serial.rxeom_len, sb->specific.serial.rxeom_char_1,
        sb->specific.serial.rxeom_char_2);
    printf("Flow Cntrl= %3X   MultiDrop Addr= %2X     MultiDrop Enabled= %s\n\r",
        sb->specific.serial.flow_ctrl, sb->specific.serial.multi_drop_addr,
        sb->specific.serial.multi_drop_enabled == 0 ? "No " : "Yes");
    printf("EOF length= %3d   EOF char      = %2X\n\r",
        sb->specific.serial.eof_length, sb->specific.serial.eof_char);
    printf("Recv Buff Avail= %s  Xmit Buff Avail=%s   Protocol Mode= %s",
        sb->specific.serial.have_recv_client_buffer == 0 ? "No " : "Yes",
        sb->specific.serial.have_xmit_client_buffer == 0 ? "No " : "Yes",
        sb->specific.serial.protocol_mode == 0 ? "Use Protocol" : "No Protocol ");
    }
    else
    if(sb->handler_type != IM_RF_BACK_PH && status != OK) err_msg(status);

if(sb->handler_type == IM_RF_BACK_PH && status == OK)  // it is the rf handler
    {
    printf("\n\rPort: %1d    Protocol Handler Ver. %s\n\r",
        port, sb->handler_ver);
    printf("\n\rRF Back Ver= %1d.%1d",
        sb->specific.rf.RF_back_major_version,
        sb->specific.rf.RF_back_minor_version);
    printf("\n\rConnected= %s",
        sb->specific.rf.net_connect_status == 0 ? "No" : "Yes");
    printf("\n\rProt Enabled= %s  Address= %d",
        sb->specific.rf.RF_protocol_enabled == 0 ? "No" : "Yes",
        sb->specific.rf.RH_device_address);
    printf("\n\rHost Addr= %d  Channel= %d  Net ID= %d",
        sb->specific.rf.RT_host_address,
        sb->specific.rf.RV_channel_select, sb->specific.rf.RW_network_ID);
    }
    else
    if(sb->handler_type == IM_RF_BACK_PH && status != OK) err_msg(status);
}

// Get the protocol's extended status for the specified port
IM_UINT  Get_Ext_Status(IM_UINT port, IM_COMM_STATUS_BUFFER_s *sb)
```

```
{
    _ES = FP_SEG((IM_COMM_STATUS_BUFFER_s far *) sb);    // pointer to buffer
    _BX = FP_OFF((IM_COMM_STATUS_BUFFER_s far *) sb);
    _CX = sizeof(IM_COMM_STATUS_BUFFER_s);    // buffer length
    _DX = port;                             // port id
    _AH = 0x71;                             // extended status function

    geninterrupt(COM_Int);

    return(_AX);
}

// Move the viewport around when spacebar is pressed
void  Move_Viewport()
{
    vp++;
    if(vp % VP_ACROSS)
        {
        VP_Move(vp_vert == 1 ? 0 : 12, vp * VP_STEPSIZE);    // right
        }
    else
        {
        VP_Move(12 * vp_vert, 0);        // up or down
        vp = 0;
        vp_vert = (vp_vert == 1 ? 0 : 1);
        }
}

// Set the video mode to 20x16
IM_UINT  Set_Video_Mode()
{
    _AH = 0x02;                             // function 2 of int 79h
    _AL = 0x03;                             // get current video mode

    geninterrupt(LCD_Int);                  // LCD display size mode manager interrupt

    vm_mode = _AL;                          // save the current video mode
    vm_params = _BX;                        // save original video parameters

    _AH = 0x02;                             // function 2 of int 79h
    _AL = 0x00;                             // set mode to 80x25

    geninterrupt(LCD_Int);                  // LCD display size mode manager interrupt

    return(_AL);                            // return status, _AL = 0 if good, 0xFF if failed
}

// Return the video mode to its original status
void    Reset_Video_Mode()
{
    _BX = vm_params;                        // restore the original video parameters
    _AH = 0x02;                             // function 2 of int 79h
    _AL = vm_mode;                          // restore the original video mode

    geninterrupt(LCD_Int);                  // LCD display size mode manager interrupt
}

// Executes the viewport move interrupt
IM_UINT  VP_Move(IM_UINT row, IM_UINT col)
{
    _AL = 0x01;                               // relative viewport move
    _AH = 0x03;                               // viewport move function
```

```
    _BX = row;
    _CX = col;

geninterrupt(LCD_Int);

    return(_AX);
}

// Display error messages
void  err_msg(IM_UINT status)
{
    switch(status) {
        case ERR_NO_HANDLER:
            printf("\n\r%s\n\r", "ERROR- No protocol handler loaded!");
            break;
        case ERR_BAD_VER:
            printf("\n\r%s\n\r", "ERROR- Incompatible revision level!");
            break;
    }
}
```

fifo.cpp

```
/* fifo.cpp
 *
 * This program demonstrates the use of the UART restore interrupt and APM
 * interface protocol.  This function is only necessary if your program is
 * communicating using the FIFO Control Register (FCR)in 16550 mode of the UART,
 *  you are not using one of the INTERMEC protocol handlers, and power management
 * is active. Since the UART restore function depends on power management, this
 * program also demonstrates the use of the PM interface.
 *
 * This program doesn't actually do any communications or set the 16550
 * mode. Instead, it illustrates how to reset the FCR (a write only register)
 * after someone turns off the reader's power.
 *
 * Pressing the 1/0 key on the Janus gives no warning to your program thru
 * power management. PM only notifies you upon power up and by then the
 * UART has been reset.
 *
 * Written: July 28, 1994
 * Last Modified: Aug. 1, 1994
 * Author: Ray Rogers
 * Compiler: Borland C++  Ver 3.1   Small Memory Model
 */

#include "dos.h"
#include "stdio.h"
#include "conio.h"

#define  PM_Int            0x15          // power mgmt interrupt
#define  TRUE              1
#define  FALSE             0
#define  NORMAL_RESUME     0x03
#define  CRITICAL_RESUME   0x04
#define  CARRY             0x01          // carry flag mask
#define  ESC               0x1b          // escape key
#define  UART_Int          0x78          // int used to control UART
#define  RESTORE           0x06          // restore function

// Function Prototypes
void  PM_Connect(void);
void  PM_Check(void);
void  PM_Disconnect(void);
void  Set_FIFO(void);
void  Store_UART(void);
void  Restore_UART(void);
int   PM_Event(void);

unsigned char  FCR;                     // FIFO ctrl register value

void  main(void)
{
    PM_Check();

    PM_Connect();

    Set_FIFO();

    Store_UART();
```

```
    while(TRUE)                          // main processing loop
      {
      if(PM_Event())
         Restore_UART();

      /*
      ** Do communications here
      */

      if(kbhit())
         if(getch() == ESC) break;     // exit program on escape

      }  // end main loop

    PM_Disconnect();
}

//APM installation check
void  PM_Check()
{
    _AX = 0x5300;
    _BX = 0x0000;

geninterrupt(PM_Int);
}

// Connect this program to Power Management
void  PM_Connect()
{
    _AX = 0x5301;                        // register as a PM partner
    _BX = 0x0000;

    geninterrupt(PM_Int);
}

// Disconnect this program from Power Management
void  PM_Disconnect()
{
    _AX = 0x5304;                         // resign as a PM partner
    _BX = 0x0000;

    geninterrupt(PM_Int);
}

// Poll PM event status
int  PM_Event()
{
int  event;

    _AX = 0x530b;                         // ask for event status

    geninterrupt(PM_Int);

    if(_FLAGS & CARRY)                    // if carry flag set function failed
        return(FALSE);

    event = _BX;                          // status returned in bx

    if(event == NORMAL_RESUME || event == CRITICAL_RESUME)
        return(TRUE);
    else
        return(FALSE);
```

```
}
// Prepare the FIFO register value
void  Set_FIFO()
{
    FCR = 0x80;                          // set FIFO trigger to 8 bytes
}

// Store the readable UART registers so that they can be restored
// if necessary
void  Store_UART()
{
    _AX = 0x0600;                        // get UART cfg and save
    _BL = 0x01;

   geninterrupt(UART_Int);

    _AX = 0x0606;                        // save FCR value
    _BL = 0x01;
    _BH = FCR;

   geninterrupt(UART_Int);

// NOTE: In a real program it would be wise to use a return value here
}

// Restore the all UART settings including FCR after power returns
void  Restore_UART()
{
    _AX = 0x0604;                        // restore UART settings
    _BL = 0x01;

   geninterrupt(UART_Int);

}
```

jbeep.cpp

```
/* jbeep.cpp
 *
 * This program demonstrates sound control using interrupts.
 * Usage is:  JBEEP volume pitch duration off_time
 *            JBEEP 3 (Loud)  1000 (Hz)  500 (ms)  20 (ms)
 *
 * Written: Aug. 20, 1993
 * Last Modified: Sept. 9, 1993
 * Author: Ray Rogers
 * Compiler: Borland C++ 3.1
*/

#include  "stdio.h"
#include  "string.h"
#include "dos.h"
#include  "conio.h"
#include  "stdlib.h"
#include  "im20lib.h"

// Defines
#define  SOUND_Int 0x7c                  // sound generation interrupt

typedef  struct                         // beeper sound structure
    {
    IM_UCHAR  volume;
    IM_UINT  pitch;
    IM_UINT  duration;
    IM_UINT  off_time;
    }    IM_BEEP_PARAMS;

// Prototypes
IM_UINT  Make_Noise(IM_UCHAR, IM_UINT, IM_UINT, IM_UINT);

void  main(int  argc, char *argv[])
{
    Make_Noise(argc > 1 ? atoi(argv[1]) : 5,        // default volume
        argc > 2 ? atoi(argv[2]) : 1000,            // default pitch
        argc > 3 ? atoi(argv[3]) : 200,             // default duration
        argc > 4 ? atoi(argv[4]) : 20);             // default off_time
}

IM_UINT  Make_Noise(IM_UCHAR volume, IM_UINT pitch, IM_UINT duration, IM_UINT off_time)
{
static  IM_BEEP_PARAMS  Beeper;          // beeper sound structure
    Beeper.volume  = volume;             // loud
    Beeper.pitch  = pitch;               // HZ
    Beeper.duration = duration;          // ms
    Beeper.off_time = off_time;          // ms
    _ES = FP_SEG((IM_BEEP_PARAMS far *) &Beeper);  // points to beeper structure
    _BX = FP_OFF((IM_BEEP_PARAMS far *) &Beeper);  // points to beeper structure

    _CX = _DX = 0;                       // to be safe
    _AX = 0x64;                          // sound function

    geninterrupt(SOUND_Int);             // generate the noise

    return(_AX);
}
```

jboot.cpp

```
/* jboot.cpp
 *
 * This program demonstrates warm boot control by using interrupts to enable/disable
 * the warm boot capability.
 *
 * Written: Aug. 25, 1993
 * Last Modified: Aug. 25, 1993
 * Author: Ray Rogers
 * Compiler: Borland C++ 3.1

#include  "stdio.h"
#include  "string.h"
#include "dos.h"
#include  "conio.h"
#include  "stdlib.h"
#include  "im20lib.h"

// Defines
#define  KEY_Int  0x16                  // keyboard interrupt
#define  STATUS  2                      // get boot status
#define  ENABLE  1                      // enable boot
#define  DISABLE  0                     // disable boot
#define  PROHIBITED  0x0000             // warm boot prohibited status

// Prototypes
IM_UINT  Set_Boot(IM_UCHAR);

void  main()
{
IM_UINT  resp;

    clrscr();
    resp = Set_Boot(STATUS);               // get the current boot status

    if(resp == PROHIBITED)                 // if boot is currently prohibited
        {
        Set_Boot(ENABLE);
        printf("%s", "\n\rWarm boot enabled...\n\r");
        }
    else
        {
        Set_Boot(DISABLE);
        printf("%s", "\n\rWarm boot disabled...\n\r");
        }

    exit(0);
}

// Sets the boot control to enable or disable
IM_UINT  Set_Boot(IM_UCHAR  status)
{
    _AH = 0xB7;                            // Ctrl-Alt-Del function
    _AL = status;
    geninterrupt(KEY_Int);

    return(_AX);
}
```

keybuff.cpp

```
/* keybuff.cpp
 *
 * This program demonstrates keyboard buffer control using interrupts.  It enables,
 * disables, and tests the expanded keyboard buffer.
 *
 * From the command line, enter:
 *   keybuff enable     or
 *   keybuff disable    or
 *   keybuff test
 *
 * Written: Sept. 1, 1993
 * Last Modified: Sept. 1, 1993
 * Author: Ray Rogers
 * Compiler: Borland C++ 3.1
*/

#include  "dos.h"
#include "stdlib.h"
#include  "string.h"
#include  "conio.h"
#include  "stdio.h"
#include  "im20lib.h"

#define  KEY_Int  0x16                  // keyboard interrupt
#define  KB_BUFF  0xb4                  // buffer enable/disable function
#define  KB_INSERT  0xb5                // insert string into buffer
#define  KB_FLUSH  0xb6                 // flush the buffer
#define  KB_OFF  0x02                   // function is off
#define  KB_ON  0x01                    // function is on
#define  KB_OK  0x00                    // function completed ok
#define  KB_NO_BUFF  0x01               // buffer is enabled
#define  KB_BUFF_FULL  0x02             // buffer is full
#define  TRUE  0x00

// Prototypes
IM_UINT  KB_Func(IM_UCHAR, IM_UCHAR);
IM_UINT  KB_Test(void);

void  main(int  argc, char  *argv[])
{
IM_UINT  resp;

    clrscr();

    if(argc == 1)                       // no command line arguments
        {
        printf("%s", "\n\rUsage: keybuff [enable | disable | test]");
        exit(0);
        }

    KB_Func(KB_FLUSH, 0);               // no effect if buffer not there

    if(TRUE == strcmp("enable", strlwr(argv[1])))
        {
        KB_Func(KB_BUFF, KB_ON);
        printf("%s", "\n\rBuffer Enabled\n\r");
        }
```

```
    if(TRUE == strcmp("disable", strlwr(argv[1])))
        {
        KB_Func(KB_BUFF, KB_OFF);
        printf("%s", "\n\rBuffer Disabled\n\r");
        }

    if(TRUE == strcmp("test", strlwr(argv[1])))
        {
        if((resp = KB_Test()) != KB_OK)
            if(resp == KB_NO_BUFF)
                printf("%s", "\n\rNo buffer installed!\n\r");
            else
                if(resp == KB_BUFF_FULL)
                    printf("%s", "\n\rBuffer is full!\n\r");
        }

    exit(0);
}

// Execute requested keyboard buffer function
IM_UINT  KB_Func(IM_UCHAR func, IM_UCHAR subfunc)
{
    _AH = func;                         // keypad function
    _AL = subfunc;                      // subfunction

    geninterrupt(KEY_Int);

    return(_AL);
}

// Test the keyboard bffer
IM_UINT  KB_Test()
{
static  char  work[] = "This is a test of the keyboard buffer string insertion function";

    _ES = FP_SEG(work);                 // address of inserted string
    _BX = FP_OFF(work);

    _CX = strlen(work);                 // length of inserted string

    _AH = KB_INSERT;                    // insert function
    _AL = 0x02;                         // ASCII mode, no scancode translation

    geninterrupt(KEY_Int);

    return(_AL);
}
```

keyclick.cpp

```
/* keyclick.cpp
 *
 * This program demonstrates key click control using interrupts. It toggles the
 * state of the key click.
 *
.* Written: Aug. 31, 1993
 * Last Modified: Sept. 1, 1993
 * Author: Ray Rogers
 * Compiler: Borland C++ 3.1
 */

#include  "dos.h"
#include "stdlib.h"
#include  "conio.h"
#include  "stdio.h"
#include  "im20lib.h"

#define  KEYPAD_Int  0x7e               // keypad interrupt
#define  KP_CLICK  0x05                 // keypad click function
#define  KP_OFF  0x00                   // function is off
#define  KP_ON    0x01                  // function is on
#define  KP_STATUS  0x02                // get status
#define  KP_ENABLED  0xff               // function is enabled
#define  VALID_SERVICE 0x00             // valid function call
#define  TRANS_OK  0x00                 // func transaction executed ok

typedef  struct  {
    IM_UCHAR  status;                   // KSCPU comm status
    IM_UCHAR  kp_status;                // 0xff if function already enabled
    IM_UINT  padding;                   // work space
    }  RESP_BUFF;

RESP_BUFF  rbuff;

// Prototypes
IM_UINT  KP_Func(IM_UCHAR, IM_UCHAR, RESP_BUFF *);

void  main()
{
    clrscr();

    if(VALID_SERVICE == KP_Func(KP_CLICK, KP_STATUS, &rbuff)) // get current status of
                                                              // click
        if(TRANS_OK == rbuff.status)
            if(rbuff.kp_status == KP_ENABLED)                 // if it is on...
                {
                KP_Func(KP_CLICK, KP_OFF, &rbuff);            // turn it off
                printf("%s", "\n\rKey click DISABLED\n\r\n\rPress a key...");
                }
            else
                {
                KP_Func(KP_CLICK, KP_ON, &rbuff);             // else, turn it on
                printf("%s", "\n\rKey click ENABLED\n\r\n\rPress a key...");
                }

    while(!kbhit());                    // wait for a keypress
    getch();
    clrscr();
    exit(0);
```

```
}
// Execute requested keypad function
IM_UINT  KP_Func(IM_UCHAR func, IM_UCHAR subfunc, RESP_BUFF *resp)
{
    _ES = FP_SEG(resp);
    _BX = FP_OFF(resp);

    _AH = func;          // keypad function
    _AL = subfunc;          // subfunction

    geninterrupt(KEYPAD_Int);

    return(_AH);
}
```

keystat.cpp

```
/* keystat.cpp
 *
 * This program demonstrates how to use the keyboard scanner and map table access using
 * interrupts.
 *
 * Written: Aug. 31, 1993
 * Last Modified: Aug. 31, 1993
 * Author: Ray Rogers
 * Compiler: Borland C++ 3.1
 */

#include  "dos.h"
#include  "stdio.h"
#include  "stdlib.h"
#include  "conio.h"
#include  "im20lib.h"

#define  KEY_Int  0x7e                  // keyboard interrupt
#define  ESC  0x1b                      // escape key
#define  FOREVER  1                     // endless loop control
#define  SCAN_VER  0x03                 // keypad scanner version function
#define  MAP_ID 0x04                    // map table ID function
#define  VALID_SERVICE  0x00            // valid function call
#define  TRANS_OK  0x00                 // function executed ok
#define  TABLE_VALID  0xff              // valid keypad table

typedef  struct  {
    IM_UCHAR  status;                   // KSCPU comm status
    IM_UCHAR  minor;                    // minor revision
    IM_UCHAR major;                     // major revision
    IM_UINT  padding;                   // work space
    } scanner_buff;                     // keypad scanner version

typedef  struct  {
    IM_UCHAR  status;                  // KSCPU comm status
    IM_UCHAR  kp_number;               // keypad number
    IM_UCHAR tabl_valid;                // table validation, 0xff = pass
    IM_UINT  chksum;                    // crc checksum
    IM_UINT  padding;                   // work space
    } map_table_buff;                   // map table ID buffer

typedef  union  {
    scanner_buff  scan;
    map_table_buff  map;
    }  RESP_BUFF;

RESP_BUFF  rbuff;                       // KSCPU response buffer

// Prototypes
IM_UINT  KP_Info(IM_UCHAR, RESP_BUFF *);

void  main()
{
    clrscr();
```

```
    if(VALID_SERVICE == KP_Info(SCAN_VER, &rbuff))
        {
        if(rbuff.scan.status == TRANS_OK)
            {
            printf("\n\rKeyPad Rev.= %d.%2d", rbuff.scan.major, rbuff.scan.minor);
            }
        }                                   // end VALID_SERVICE

    if(VALID_SERVICE == KP_Info(MAP_ID, &rbuff))
        {
        if(rbuff.map.status == TRANS_OK)
            {
            printf("\n\rMap ID= %X", rbuff.map.kp_number);
            printf("\n\rValid Table= %s",
                 rbuff.map.tabl_valid == TABLE_VALID ? "Yes" : "No ");
            printf("\n\rCheck sum= %4X", rbuff.map.chksum);
            }
        }                                   // end VALID_SERVICE

    printf("%s", "\n\r\n\rPress any key...");
    while(!kbhit());                        // wait for key press
    getch();
    clrscr();

    exit(0);
}

// Do keypad functions
IM_UINT  KP_Info(IM_UCHAR func, RESP_BUFF *rbuff)
{
    _ES = FP_SEG((RESP_BUFF *) rbuff);   // far pointer to response buffer
    _BX = FP_OFF((RESP_BUFF *) rbuff);   // far pointer to response buffer

    _AH = func;                          // scanner version or map ID function

    geninterrupt(KEY_Int);

    return(_AH);
}
```

numpad.cpp

```
/* numpad.cpp
 *
 * This program demonstrates key click control using interrupts.  It toggles the
 * modality of the keypad.
 *
 * Written: Sept. 1, 1993
 * Last Modified: Sept. 1, 1993
 * Author: Ray Rogers
 * Compiler: Borland C++ 3.1
 */

#include  "dos.h"
#include "stdlib.h"
#include  "conio.h"
#include  "stdio.h"
#include  "im20lib.h"

#define  KEYPAD_Int  0x7e               // keypad interrupt
#define  KP_NUMPAD  0x06                // keypad numpad function
#define  KP_OFF  0x00                   // function is off
#define  KP_ON_NUM_ON 0x01              // pad enabled, numlock is on
#define  KP_ON_NUM_OFF 0x02             // pad enabled, numlock is off
#define  KP_STATUS  0x03                // get status
#define  KP_ENABLED  0xff               // function is enabled
#define  VALID_SERVICE 0x00             // valid function call
#define  TRANS_OK  0x00                 // func transaction executed ok
#define  ESC    0x1b                    // ESC key
#define  FOREVER  1                     // endless loop

typedef  struct  {
    IM_UCHAR  status;                   // KSCPU comm status
    IM_UCHAR  pad_status;               // 0xff if function already enabled
    IM_UCHAR  permit_status;            // 0xff if switching modality from keypad
                                        // permitted
    IM_UINT  padding;                   // work space
    } RESP_BUFF;

RESP_BUFF  rbuff;

// Prototypes
IM_UINT  KP_Func(IM_UCHAR, IM_UCHAR, RESP_BUFF *);

void  main()
{
IM_UCHAR  ch;

    clrscr();

    printf("%s", "\n\rCurrent NumPad Status:");

    if(VALID_SERVICE == KP_Func(KP_NUMPAD, KP_STATUS, &rbuff)) // get curr. status of
                                                               //click
        if(TRANS_OK == rbuff.status)
            if(rbuff.pad_status == KP_ENABLED)  // if it is on...
                printf("%s", "\n\rNUMPAD ENABLED - \n\rPC KB 102 codes in use\n\r\n\r");
            else
                printf("%s", "\n\rNUMPAD DISABLED - \n\rQWERTY codes in use\n\r\n\r");
                printf("%s", "Choose New Setting:\n\r1. Disable Numpad\n”);
                printf("%s", "\r2. Enable, NumLk On\n\r3. Enable, NumLk Off\n\r");
```

```
    KP_Func(KP_NUMPAD, KP_OFF, &rbuff);  // make sure we can get menu choice

    ch = ' ';
    while(ch != ESC)
        {
        ch = getch();

        switch(ch)
            {
                case  '1':
                    KP_Func(KP_NUMPAD, KP_OFF, &rbuff);
                    ch = ESC;
                    break;
                case  '2':
                    KP_Func(KP_NUMPAD, KP_ON_NUM_ON, &rbuff);
                    ch = ESC;
                    break;
                case  '3':
                    KP_Func(KP_NUMPAD, KP_ON_NUM_OFF, &rbuff);
                    ch = ESC;
                    break;
            };
        }

    clrscr();
    exit(0);
}

// Execute requested keypad function
IM_UINT  KP_Func(IM_UCHAR func, IM_UCHAR subfunc, RESP_BUFF *resp)
{
    _ES = FP_SEG(resp);
    _BX = FP_OFF(resp);

    _AH = func;                          // keypad function
    _AL = subfunc;                       // subfunction
    _CL = 0x01;                          // keypad mode switching permitted

    geninterrupt(KEYPAD_Int);

    return(_AH);
}
```

open.cpp

```
/* open.cpp
 *
 * This program demonstrates how to open COM2 on the 2D JANUS.
 *
 * Written: Feb. 10, 1999
 * Last Modified: Feb. 10, 1999
 * Author: Peter J. Girodat
 * Compiler: Borland C++ 3.1
 */

#include <dos.h>
#include <stdio.h>
#include <stdlib.h>

#define RS_VECTOR          0x7D      /* Interrupt 7DH Reader Services      */
#define RS_ASIC_RESET      0x0C      /* Function  0CH ASIC Reset           */
#define RS_CLOSE_COM2      0x0D      /* Function  0DH Close COM2           */
#define RS_ALL_SUCESS      0x0100    /* Reader Service sucessful           */
#define RS_INVALID_REQUEST 0x8100    /* Invalid function Hex code received */

void main()
{
   _AH = RS_ASIC_RESET;             // AH = Function 0CH
   _AL = 0;

   geninterrupt(RS_VECTOR);

   if (_AX == RS_ALL_SUCESS)
      printf ("COM2 OPEN\n");
   else if (_AX == RS_INVALID_REQUEST)
      printf ("JANUS 1D Reader\n");
}
```

phterm.cpp

```
/* phterm.cpp
 *
 * This program demonstrates how to use interrupts and protocol.  It requires a
 * Janus running PHIMEC.EXE and a connection to something else using an INTERMEC
 * protocol.
 *
 * Written: Aug. 12, 1993 as PHCOMM, derived PHTERM Feb. 28, 1994
 * Last Modified: March 1, 1994
 * Author: Ray Rogers
 * Compiler: Borland C++ 3.1,  small model
 */

#include  "stdio.h"
#include  "string.h"
#include "dos.h"
#include  "conio.h"
#include  "stdlib.h"
#include  "im20lib.h"

// Defines
#define  COM_Int  0x14                  // Communications interrupt
#define  LCD_Int  0x79                  // LCD size mode manager interrupt
#define  SOUND_Int  0x7c                // sound generation interrupt
#define  VIDEO_Int  0x10                // video interrupt
#define  ESC  0x1b                      // char 27,  Escape
#define  TXBUFFSIZE  512                // transmit buffer size
#define  RXBUFFSIZE  512                // receive buffer size
#define  COM_PORT  3                    // default to com port 0 for Janus
#define  LOUD  3                        // loud volume for beep
#define  NORMAL  2                      // normal volume for beep
#define  LOW  1                         // low volume for beep
#define  BOOL  IM_UINT
#define  TRUE  1
#define  FOREVER  TRUE

typedef  struct                         // beeper sound structure
    {
    IM_UCHAR  volume;
    IM_UINT  pitch;
    IM_UINT  duration;
    IM_UINT  off_time;
    }  IM_BEEP_PARAMS;

    // struct for window control on JANUS
    typedef  struct
    {
    IM_UCHAR  ur;                       // upper row
    IM_UCHAR  uc;                       // upper column
    IM_UCHAR  lr;                       // lower row
    IM_UCHAR  lc;                       // lower column
    IM_UCHAR  cr;                       // current row, relative to this window
    IM_UCHAR  cc;                       // current column, relative to this window
    }  JANUS_WINDOW;

// Global Variables
IM_UINT  vm_mode;                       // saves original video mode
IM_UINT  vm_params;                     // saves original video parameters
IM_COM_DATA_BUFFER  s_struct;           // send structure
IM_COM_DATA_BUFFER  r_struct;           // receive structure
```

```
IM_UCHAR  s_buff[TXBUFFSIZE];           // send buffer
IM_UCHAR  r_buff[RXBUFFSIZE];           // receive buffer
JANUS_WINDOW  iw = {0, 0, 0, 19, 0, 0};  // input window
JANUS_WINDOW  ow = {3, 0, 14, 19, 0, 0}; // output window

// Prototypes
IM_UINT  Ck_RX_Buffer_Status(IM_UINT portid);
IM_UINT  TX_Buffer(IM_UINT portid);
IM_UINT  RX_Buffer(IM_UINT portid);
IM_UINT  Set_Video_Mode(void);
IM_UINT  Make_Noise(IM_UCHAR, IM_UINT, IM_UINT, IM_UINT);
IM_UINT  Get_Input(void);
void  Reset_Video_Mode(void);
void  Cancel_TX(IM_UINT port);
void  Cancel_RX(IM_UINT port);
void  Pos_Cursor(JANUS_WINDOW *, IM_UINT, IM_UINT);
void  Shift_Left(IM_UINT);
void  Blank_Input(void);
void  Ck_y(JANUS_WINDOW *);

void  main()
{
    if(Set_Video_Mode())                // set video mode, sound beep and exit on failure
        {
        Make_Noise(LOUD, 2000, 200, 20);  // loud, high pitch
        Make_Noise(NORMAL, 500, 250, 20);  // normal, low pitch
        exit(1);
        }

    Pos_Cursor(&ow, 0, -1);
    printf("%s", "_____________________");
    ow.cr = 0;
    Pos_Cursor(&iw, 0, 0);

    r_buff[0] = '\0';
    RX_Buffer(COM_PORT);                // read the comm line

    // main processing loop, exit loop on ESC key
    while(FOREVER)
        {
        if(Get_Input() == ESC)          // get input from keyboard or comm line
            break;

            TX_Buffer(COM_PORT);
        }                               // end main loop

    Reset_Video_Mode();                 // return video mode to original settings
    Cancel_TX(COM_PORT);                // shut down interface to protocol handler
    Cancel_RX(COM_PORT);

    Make_Noise(LOUD, 1000, 200, 20);    // loud, high pitch
    Make_Noise(NORMAL, 250, 250, 20);   // normal, low pitch

    exit(0);
}

// Prepare input line for new input by clearing out the old
void  Blank_Input()
{
    Pos_Cursor(&iw, 0, 0);
    printf("%s", "                    ");
    Pos_Cursor(&iw, 0, 0);
```

```
}

// Cancel the RX buffer
void  Cancel_RX(IM_UINT port)
{
    _DX = port;
    _AH = 0x84;

    geninterrupt(COM_Int);
}

// Cancel the TX buffer
void  Cancel_TX(IM_UINT port)
{
    _DX = port;
    _AH = 0x85;

    geninterrupt(COM_Int);
}

//Check the receive buffer is a misnomer for this function since it actually instructs
//the protocol handler to receive a packet.  After that, you must check
//protocol_xfer_status periodically to see if anything has been received yet.
IM_UINT  Ck_RX_Buffer_Status(IM_UINT port)
{
   r_struct.command = IM_CU_RECV_AGAIN;          // set the cmd to receive
    *(r_struct.data_ptr) = '\0';
   r_struct.comm_length = 0;

    _DX = port;                          // select the com port
    _AH = 0x17;                          // check com port receive buffer status function

    geninterrupt(COM_Int);

    return(r_struct.protocol_xfer_status);
}

// Check the row position that is about to be used for displaying a line
// received from the comm port.  If it is the bottom of the screen, scroll
// the window to make room.
void  Ck_y(JANUS_WINDOW *jw)
{
    if(jw->ur+jw->cr > jw->lr)
        {
        jw->cr--;
        _AH = 0x06;                      // video scroll function
        _AL = 1;                         // number of lines to scroll
        _CH = jw->ur;                    // upper left row
        _CL = jw->uc;                    // upper left column
        _DH = jw->lr;                    // lower right row
        _DL = jw->lc;                    // lower right col
        _BH = 0x07;                      // normal video mode

        geninterrupt(VIDEO_Int);
        }
}

// Position the cursor outside the defined text window
void  Pos_Cursor(JANUS_WINDOW *jw, IM_UINT loc, IM_UINT row)
{
    jw->cr = row;
    jw->cc = loc;
```

```
    _DH = jw->ur+row;          // row
    _DL = jw->uc+loc;          // column
    _BH = 0;           // page
    _AH = 0x02;          // position cursor function
    geninterrupt(VIDEO_Int);    // video
}

// Get the next input string from the keyboard while watching for input from the comm line
IM_UINT  Get_Input()
{
IM_UINT  x, ch;

    ch = x = 0;
    iw.cc = 0;
    while(ch != ESC && ch != '\r')       // keybd input ends with CR or ESC
        {
        if(kbhit())
            {
            ch = getch();                // use getch(), then printf() because JANUS

            if(ch == '\r')
                Blank_Input();           // clears input line for clean start
            else
                if(iw.uc+iw.cc >= iw.lc)
                   Shift_Left(x);        // input exceeds line length, scroll left
                else
                   Pos_Cursor(&iw, iw.cc, 0);

            printf("%c", ch);            // doesn't like getche() or cputs()
            if(ch == '\b')               // backspace is the only supported editing key
                {
                x--;
                iw.cc--;
                }
            else
                {
                iw.cc++;
                s_buff[x++] = ch;
                }
            }

        if(r_struct.protocol_xfer_status == 0x0601)  // 0x0601 is comm buffer done from
                                                     // last recv request
            {
            *(r_struct.data_ptr + r_struct.comm_length) = '\0'; // add null to receive
            Ck_y(&ow);                                          // buffer
            Pos_Cursor(&ow, 0, ow.cr);
            ow.cr++;
            printf("%s\n\r", r_buff);                // print buffer contents
            Pos_Cursor(&iw, 0, 0);
            Ck_RX_Buffer_Status(COM_PORT);           // issue request for next packet
            }
        }

    strcpy(&s_buff[x], "\n");            // add a line feed and a null character
    return(ch);
}

// read the receive buffer
IM_UINT  RX_Buffer(IM_UINT port)
{
    r_struct.command = IM_CU_RECEIVE;    // set the cmd to receive
```

```
    r_struct.user_length = RXBUFFSIZE;
    r_struct.data_ptr = (IM_UCHAR far *) r_buff;
    r_struct.protocol_mode = IM_USE_PROTOCOL;
    _ES = FP_SEG((IM_COM_DATA_BUFFER far *) &r_struct);
    _BX = FP_OFF((IM_COM_DATA_BUFFER far *) &r_struct);

    _DX = port;                          // select the com port
    _AH = 0x60;                          // receive buffer

    geninterrupt(COM_Int);

    return(r_struct.protocol_xfer_status);
}

// transmit the transmit buffer
IM_UINT  TX_Buffer(IM_UINT port)
{
    s_struct.command = IM_CU_TRANSMIT;   // set the cmd to transmit
    s_struct.data_ptr = (IM_UCHAR far *) s_buff;
    s_struct.user_length = strlen((IM_UCHAR far *) s_buff);
    s_struct.protocol_mode = IM_USE_PROTOCOL;

    _ES = FP_SEG((IM_COM_DATA_BUFFER far *) &s_struct);
    _BX = FP_OFF((IM_COM_DATA_BUFFER far *) &s_struct);

    _DX = port;                          // select the com port
    _AH = 0x50;                          // transmit buffer

    geninterrupt(COM_Int);

    return(s_struct.protocol_xfer_status);
}

// Make a noise to indicate a failure condition of some kind has occurred
IM_UINT  Make_Noise(IM_UCHAR volume, IM_UINT pitch, IM_UINT duration, IM_UINT off_time)
{
static  IM_BEEP_PARAMS  Beeper;          // beeper sound structure
    Beeper.volume  = volume;             // loud
    Beeper.pitch  = pitch;               // HZ
    Beeper.duration = duration;          // ms
    Beeper.off_time = off_time;          // ms

    _ES = FP_SEG((IM_BEEP_PARAMS far *) &Beeper);    // points to beeper structure
    _BX = FP_OFF((IM_BEEP_PARAMS far *) &Beeper);    // points to beeper structure

    _CX = _DX = 0;                        // to be safe
    _AX = 0x64;                           // sound function

    geninterrupt(SOUND_Int);              // generate the noise

    return(_AX);
}

// Set the video mode to 20x16
IM_UINT  Set_Video_Mode()
{    _AH = 0x02;                         // function 2 of int 79h
    _AL = 0x03;                          // get current video mode

    geninterrupt(LCD_Int);               // LCD display size mode manager interrupt

    vm_mode = _AH;                       // save the current video mode
    vm_params = _BX;                     // save original video parameters
```

```
    _AH = 0x02;                         // function 2 of int 79h
    _AL = 0x01;                         // set mode to 20x16

    geninterrupt(LCD_Int);              // LCD display size mode manager interrupt

    return(_AL);                        // return status, _AL = 0 if good, 0xFF if failed
}

// Return the video mode to its original status
void  Reset_Video_Mode()
{
    _BX = vm_params;                     // restore the original video parameters
    _AH = 0x02;                          // function 2 of int 79h
    _AL = 0x00;
    _BH = vm_mode;                      // restore the original video mode

    geninterrupt(LCD_Int);              // LCD display size mode manager interrupt
}

// When input exceeds line length, scroll the line left
void  Shift_Left(IM_UINT x)
{
    s_buff[x] = ' ';
    s_buff[x+1] = '\0';
    Pos_Cursor(&iw, 0, 0);
    printf("%s", s_buff+x-iw.lc+1);
    Pos_Cursor(&iw, iw.lc-1, 0);
}
```

pl220ver.cpp

```
 /* pl220ver.cpp
 *
 * This program demonstrates how to display the Symbol PL220 Decoder
 * firmware revision level using interrupts.
 *
 * Written: Jan. 11, 1999
 * Last Modified: Feb. 25, 1999
 * Author: Peter J. Girodat
 * Compiler: Borland C++ 3.1
 */

#include <dos.h>
#include <stdio.h>
#include <stdlib.h>

#define RS_VECTOR          0x7D      /* Interrupt 7DH Reader Services */
#define RS_PL220_VER       0x0A      /* Function  0AH PL 220 Version  */
#define RS_ALL_SUCESS      0x0100    /* Reader Service sucessful           */
#define RS_INVALID_REQUEST 0x8100    /* Invalid function Hex code received */

void main()
{
   char pl220_version[16];

   _ES = FP_SEG((void*)pl220_version); // ES:BX Far pointer to pl220_version
   _BX = FP_OFF((void*)pl220_version);
   _AH = RS_PL220_VER;                 // AH = Function 0AH
   _AL = 0;

   geninterrupt(RS_VECTOR);

   if (_AX == RS_ALL_SUCESS)
      printf("%s\n",pl220_version);
   else if (_AX == RS_INVALID_REQUEST)
      printf ("JANUS 1D Reader\n");
}
```

vp_cur.cpp

```
/* vp_cur.cpp
.*
.* This program demonstrates how to enable and disable the viewport follows cursor
 * function.
 *
 * Written: Sept. 23, 1993
 * Last Modified: September 23, 1993
 * Author: Ray Rogers
 * Everett, WA.
.* Borland C++ 3.1, small model
 */

#include "string.h"
#include "dos.h"
#include "stdio.h"
#include "im20lib.h"

#define KEY_Int  0x7e
#define OFF  0x11                       //vp does not follow cursor
#define ON  0x10                        //vp follows cursor

//Prototypes
IM_UINT  VP_Set(IM_UINT);

void main(int argc, char *argv[])
{
    if(argc > 1)
        {
        if(strcmp(strupr(argv[1], "OFF") == 0)
            VP_Set(OFF);
        else
            VP_Set(ON);          //default is vp follows
        }
    else
        printf("\n\rUsage: vp_cur[on | off]\n\r");
}

//enable/disable viewport follows cursor
IM_UINT  VP_Set(IM_UINT func)
{
    _AH = func;
    geninterrupt(KETY_Int);
    return(_AX);
}
```

I

Index

A

B

C

D

E

F, H

I

J, K, L

M

N, O

P

R

S

T

U, V, W