Palminomicon

Palm OS Overview

Terminology

Hotsync

The act of synchronizing the data stored on the Palm with a backup stored on a PC. This can be done through the cradle or over a modem or infrared connection.

Cradle

Where the palm is docked for a hotsync.

Beaming

The act of transferring an object (binary file, database, database record, etc.) from one device to another over the IR port.

Handspring Visor

A less expensive Palm clone.

Development

There are a suite of tools for Linux, including a GCC cross compiler.

Codewarrior released an IDE and emulator for Windows.

There is a Palm emulator called "POSE" that runs under Linux and Windows.

Processor

All Palm OS machines run a clone of the 68000 processor running at 16MHz, called Dragonball (MC68328) on the Palms. There is also a more modern Dragonball EZ processor (MC68EZ328) in use in many models.

Display

Screen resolution is 160x160.

In PalmOS 3.x and above, 4 shades of gray are supported, though most apps operate in monochrome mode.

(Some hacks hit 16 shades of gray using lots of inline assembly.)

The Palm IIIc is the only device in the family that shows color, and it is 8-bit indexed, 8-bits per channel.

Operating System

Starting with the Palm III, the OS version is 3.0. This is the first version that supports grayscale and a variety of other features. It is also the lowest version we will support and the version we are building for.

In almost all cases, OS versions are backwards binary compatible. The API might change slightly from version to version.

Other PalmOS devices run various versions including 3.0, 3.0.2, 3.1, 3.2, 3.3, and 3.5.

Version 3.5 is required for color applications.

Memory Management

Most devices come with 2MB, 4MB or 8MB of RAM/storage.

Each application running has access to the dynamic heap, which is quite small. Older devices have a 96K heap, while the newer ones have 128K.

The memory manager handles "chunks" of data which it is free to move around.

Chunks can be locked down for use by a program.

Databases

Databases are a special form of memory chunks that are tagged with identifying information that can associate them with an application.

Databases can be thought of as memory mapped "files" which exist in RAM.

Sample usage: MemoPad.

Records can be added to and deleted from a database, and databases can be stored sorted.

Databases also have an area for application-specific information.

For ease, a file streaming interface has been implemented. Many apps continue to use the Database interface since it works well with their functionality.

Databases generally have the ".PDB" extension, but that varies according to specialized database types.

Databases can be prebuilt and

Applications

Palm OS supports multiple tasks (processes) in theory, but not in practice.

The Launcher starts a new process for an Application, but the Application generally doesn't continue to run in the background when another Application is launched.

Each Application generally contains a "Form" which is a Window containing widget. (The application is free to create as many Windows as it wants, and they need not be Forms.)

The Application runs as an event-driven window system. Events are passed from the OS through a series of event handlers, some of which are within the application. The app can choose to handle certain events.

The executable is a ".PRC" file, which is itself a Database. It includes the binary code for the system as well as the resources used.

Resource files

Resource files are specialized Databases that hold information about the UI widgets used in Forms and Windows, as well as other misc information about the Application (e.g. version number).

The resource file is defined as a text file (Unix) or using a GUI design tool (Codewarrior) and is compiled using a "resource compiler" into a format that can be included in the executable binary.

Widgets can be added and removed programatically at runtime, but it's generally a PITA compared to using the resource compiler.

Communication

Serial communication is supported through the cradle, and over the IR port using a protocol called IRCOMM.

The Palm VII supports wireless internet connections.

Other devices support serial modem connections.

Some cradles use a USB connection for faster syncs, but the throughput is limited by the speed of the processor to somewhere under 4MB.

The IR port uses a common standard protocol called IrDA. It's a layered protocol where the upper layers correspond to high level protocols such as IRCOMM and IrOBEX.

IrOBEX is the protocol used when "beaming" data from one device to another.

The next layers down in the IR protocol stack can be accessed for complete control of the data transmitted (e.g. for a game.) (A homebuilt library for simplifying this complex process into a few function calls is being written.)

Drawing Primitives

In Palm OSs before 3.5, there are black and white drawing primitives (line, rectangle, point, etc.) and hashed fill patterns.

Palm OS 3.5 introduces grayscale and color drawing primitives. (Which we can't use unless we also code support for older 3.0 models.)

Applications can create an offscreen window for drawing and double-buffering.

Bitmaps

Bitmaps can be monochrome, grayscale, or color.

Bitmaps can be dynamically constructed at runtime using our homebuilt bitmap library.

Support for modification of bitmaps in versions before 3.5 is virtually nonexistant (except with our bitmap lib.)

Palm OS 3.5 introduces a method for using drawing primitives directly on a bitmap. (Which we can't use.)

Misc Hardware

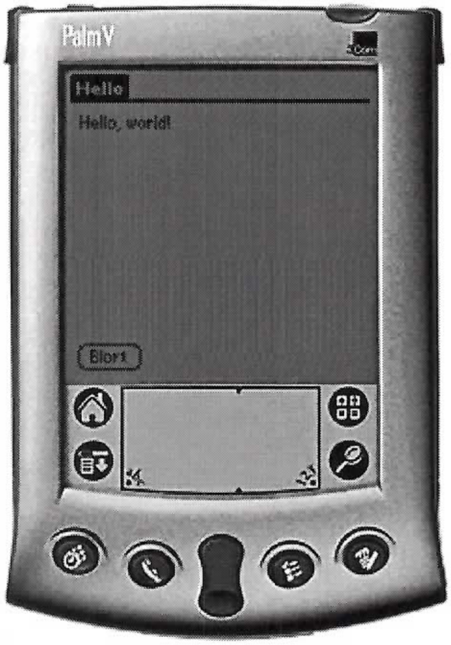
Some Palms, such as the IIIc and V, have built in rechargable batteries which power up while the device is docked in the cradle.

Linux (recent development kernels) supports IrDA, and we possess a cool IR dongle to experiment with transmitting from a PC to a palm.

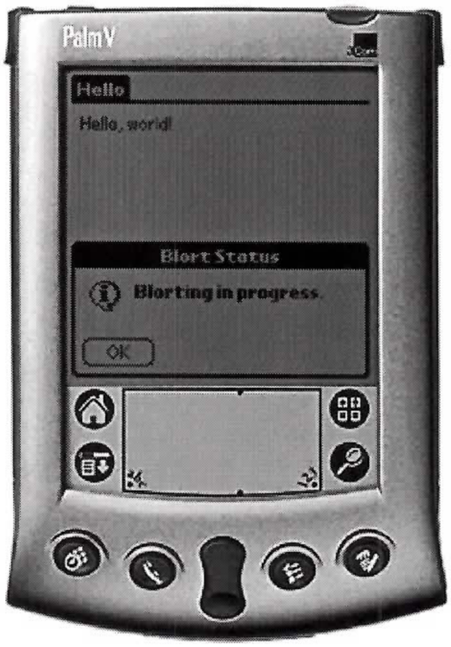
The Visor has a plug-in slot for "Springboard Modules" ("Springboard" being the name of the tech used for the slot.) Examples of these cards are video games, memory expansion, cameras, MP3 players, GPSs (not yet available), etc.

■

Resources



Hello, world!



Hello, world!

Blorting in progress.

`Hello.rcp`:

```
#include "HelloRsc.h"

/*
** Version number for application
*/
VERSION ID 1000 "1.0"

/*
** Description of main form
*/
FORM ID HelloForm AT (0 0 160 160)
BEGIN
        TITLE "Hello"
        BUTTON "Blort" ID BlortButton AT (5 BOTTOM@156 AUTO AUTO)
END

/*
** An alert message box
*/
ALERT ID BlortAlert
INFORMATION
BEGIN
        TITLE "Blort Status"
        MESSAGE "Blorting in progress."
        BUTTONS "OK"
END
```

```
/*
** Hello.c
*/

#include <Pilot.h>
#include "HelloRsc.h"
#ifdef __GNUC__
#include "Callbacks.h"
#endif

/*
** StartApplication()
*/
static Err StartApplication(void)
{
    // when the application starts, load its main form and jump to it
    FrmGotoForm(HelloForm);

    return 0;
}

/*
** StopApplication()
*/
static void StopApplication(void)
{
    // any cleanup code goes here
}

/*
** FormHandleEvent()
*/
static Boolean FormHandleEvent(EventPtr event)
{
    // handled is set true if no more processing on this event is required
    Boolean handled = false;

#ifdef __GNUC__
    CALLBACK_PROLOGUE
#endif

    // what kind of event is it?
    switch (event->eType) {

        // the form is just opening, so init stuff and draw it
        case frmOpenEvent:
            FrmDrawForm(FrmGetActiveForm()); // display the form
            WinDrawChars("Hello, world!", 13, 5, 20);
            handled = true;
            break;

        // a button has been pressed
        case ctlSelectEvent:
            switch(event->data.ctlSelect.controlID) {
                case BlortButton:
                    FrmAlert(BlortAlert); // pop up the alert box
                    handled = true;
                    break;
            }
            break;
    }

#ifdef __GNUC__
    CALLBACK_EPILOGUE
#endif

    return handled;
}

/*
** ApplicationHandleEvent()
*/
static Boolean ApplicationHandleEvent(EventPtr event)
{
    FormPtr frm;
    Int formId;
    Boolean handled = false;

    // this event happens when the form is first loaded and displayed
    if (event->eType == frmLoadEvent) {
        formId = event->data.frmLoad.formID;
        frm = FrmInitForm(formId); // init
        FrmSetActiveForm(frm); // activate

        switch (formId) {
            case HelloForm:
                // set the event handler for this form
                FrmSetEventHandler(frm, FormHandleEvent);
                break;
        }
        handled = true;
    }

    return handled;
}

/*
** EventLoop()
*/
static void EventLoop(void)
{
    EventType event;
    Word error;

    do {
        EvtGetEvent(&event, evtWaitForever);

        if (!SysHandleEvent(&event)) {
            if (!MenuHandleEvent(NULL, &event, &error)) {
                if (!ApplicationHandleEvent(&event)) {
                    FrmDispatchEvent(&event);
                }
            }
        }
    } while (event.eType != appStopEvent);
}


/*
** PilotMain()
*/
DWord PilotMain(Word cmd, Ptr cmdPBP, Word launchFlags)
{
    Err err = 0;

    if (cmd == sysAppLaunchCmdNormalLaunch) {
        if ((err = StartApplication()) == 0) {
            EventLoop();
            StopApplication();
        }
    }
    return err;
}
```

HelloRsc.h

```
/* pilrc generated file.  Do not edit!*/
#define BlortAlert 9997
#define BlortButton 9998
#define HelloForm 9999
```

```
#ifndef __CALLBACK_H__
#define __CALLBACK_H__

/* This is a workaround for a bug in the current version of gcc:
   gcc assumes that no one will touch %a4 after it is set up in crt0.o.
   This isn't true if a function is called as a callback by something
   that wasn't compiled by gcc (like FrmCloseAllForms()).  It may also
   not be true if it is used as a callback by something in a different
   shared library.
   We really want a function attribute "callback" which will insert this
   progloue and epilogoue automatically.
- Ian */

register void *reg_a4 asm("%a4");

#define CALLBACK_PROLOGUE \
   void *save_a4 = reg_a4; asm("move.l %%a5,%%a4; sub.l #edata,%%a4" : :);
#define CALLBACK_EPILOGUE reg_a4 = save_a4;

#endif
```

Palm OS®

Home ▸
Palm OS Platform ▸
Palm Economy ▸
Licensing ▸
OEM Partnerships ▸
Business Alliances ▸
Enterprise ▸
Developers
Getting Started ▸
Technical Resources
Business and Marketing ▸
Solution Provider Program ▸
Platinum Program ▸
News and Events ▸
Provider Pavilion ▸

About Palm, Inc. ▸
Site Map ▸
Contacts ▸

Quick Index Go
Business Info
Conduits
Contact Palm
Creator ID
Dev Seeding

Search GO ▸

▸ New Developers Technologies ▸ Tools
▸ Knowledge Base ▸ Documentation ▸ Support

Hardware Comparison Matrix

These are the currently released Palm OS® platform devices, including those from Palm Inc. licensees and OEMs. This list of the major differences between products is designed help developers understand product differences and product histories. This page is not intended to be a complete list of differences between products; minor differences in hardware and/or ROM revisions may also exist.

Palm Inc. Products	Total RAM	Dynamic Heap	Palm OS® Version	CPU Type	TCP Support	IR Support	Flashable ROM	
Pilot 1000	128K	32K	1.0	DB				No backlight.
Pilot 5000	512K	32K	1.0	DB				No backlight.
Pilot w/1MB upgrade	1MB	32K	1.0	DB				No backlight.
PalmPilot[tm] Personal	512K	32K	2.0	DB				
PalmPilot[tm] Pro	1MB	64K	2.0	DB	x			
PalmPilot[tm] Pro Upgrade	1MB	64K	2.0	DB	x			Upgrade for Pilot 1000, Pilot 5000, and PalmPilot Personal
Palm III[tm] organizer	2MB	96K	3.0	DB	x	x	x	
Palm[tm] 2MB upgrade	2MB	96K	3.0	DB	x	x	x	Upgrade for Pilot 1000, Pilot 5000, and PalmPilot Personal. Adds flash and IR.
Palm IIIc[tm] organizer	8MB	128K	3.5	EZ	x	x	x	Rechargable lithium ion batteries. Color 160x160 screen.
Palm IIIx[tm] organizer	4MB	128K	3.1	EZ	x	x	x	
Palm IIIe[tm] organizer Special Edition	2MB	128K	3.3	EZ	x	x		French & German
Palm IIIe[tm] organizer	2MB	128K	3.3	EZ	x	x		Spanish
Palm IIIe[tm] organizer & Special Edition	2MB	128K	3.1	EZ	x	x		
Palm IIIxe[tm] organizer	8MB	128K	3.5	EZ	x	x	x	Rechargable lithium ion batteries
Palm V[tm] organizer	2MB	128K	3.1	EZ	x	x	x	Rechargable lithium ion batteries
Palm VII[tm] organizer	2MB	128K	3.2	DB§	x	x	x	Built-in wireless connections.
Palm Vx[tm] organizer	8MB	128K	3.3/3.5	EZ	x	x	x	Rechargable lithium ion batteries. 2MB ROM
Licensee and OEM Devices								
PageMart Synapse PagerCard	2MB	96K	3.0@	DB	x		x	Includes alpha pager.
Symbol SPT1500	2MB	96K	3.0.2¶	DB	x	x	x	Includes built-in scanner
Symbol SPT1700	2 or 8MB	96K	3.2	DB	x	x	x	Ruggedized device with built-in scanner.
Symbol SPT1740	2 or 8MB	96K	3.2	DB	x	x	x	Ruggedized device with built-in scanner and Spectrum 24 wireless radio network

								interface.
Handspring Visor	2 or 8MB	128K	3.1	EZ	x	x		Springboard[tm] slot and built-in microphone.
IBM WorkPad PC Companion	2MB	96K	3.0	DB	x	x	x	Original WorkPad PC Companion
IBM WorkPad PC Companion	4MB	128K	3.1	EZ	x	x	x	English and Japanese versions.
IBM WorkPad c3 PC Companion	2MB	128K	3.1	EZ	x	x	x	English and Japanese versions. Rechargable lithium ion batteries. Same form factor as Palm V[tm] organizer.
Qualcomm pdQ smartphone	2MB	128K	3.02	DB	x	x		Built-in phone - pdQ 800 and pdQ 1900 differ only in carrier frequency -- a difference necessary based on the area of use.
TRG TRGpro	8MB	128K	3.3	EZ	x	x	x	CompactFlash[tm] Type I/II slot. Improved audio.

CPU Types: If listed as "DB", it is the Motorola Dragonball NC68328 chip. If CPU is listed as "EZ", it is the Motorola Dragonball EZ chip, officially called the Motorola NC68EZ328.

§ Current Palm VII[tm] devices have the original Dragonball CPU. Future devices may use the Dragonball EZ CPU.

¶ Symbol SPT1500 ROM version should be 3.0.2r3. There may be devices with earlier ROM versions in the channel. Such units should update to ROM version 3.0.2 RELEASE 3 which fixes a potential data loss problem involving low-battery power management.

@ PagerCard does not add IR capability to the Palm Pilot 1000, 5000, Personal, or Pro models. However, it does not affect the existing IR capability of a Palm III[tm] device.

Hardware families

The following groups of devices use identical logic board designs, except for the memory card::

1. Pilot 1000, Pilot 5000, Pilot w/1MB upgrade
2. PalmPilot Personal, PalmPilot Professional.
3. Palm III[tm] organizer, Palm III[tm] Upgrade, original IBM WorkPad PC Companion
4. Palm V[tm] organizer, IBM Workpad c3 PC companion

Palm OS®

Home ▸
Palm OS Platform ▸
Palm Economy ▸
Licensing ▸
OEM Partnerships ▸
Business Alliances ▸
Enterprise ▸
Developers ▾
Getting Started ▸
Technical Resources ▸
Business and Marketing ▸
Solution Provider Program ▸
Platinum Program ▸
News and Events ▸
Provider Pavilion ●

About Palm, Inc. ▸
Site Map ▸
Contacts ▸

Quick Index Go
Business Info
Conduits
Contact Palm
Creator ID
Dev Seeding

Search GO ▸

ROM Image File Downloads (Clickwrap - USA Developers Only)

Each downloadable package contains a text file entitled "Appendix". Each such appendix shall be considered an appendix to your Prototype License and Confidentiality Agreement.

Palm OS® ROMs are intended for use with the Palm OS Emulator. For general information about the Emulator and links to documentation, tools, third-party extras, and other Emulator-related resources, see the Palm OS Emulator page. The newest versions of the Emulator are found on the Emulator Seeding page.

These images are not intended for reflashing ROMs in actual devices. This is **not supported.** If you do it anyway, make sure that the ROM image file matches the device. Doing this incorrectly can damage your device and/or your palm.net account. Since it is not supported, neither Palm customer support nor development support teams can facilitate nor help you recover from reflashing operations.

About ROM Types

For Palm OS software versions before 3.5, we provide individual ROM files for each device. Starting with Palm OS software version 3.5, we post platform ROMs for each ROM type. These are very nearly identical to shipping device ROMs for Palm OS 3.5 and greater and should be sufficient for any debugging purpose. However, if you wish to upload a ROM image from a device you posess, see the Emulator documentation for details.

EZ	*Devices with CPU MC68EZ328*	*Palm[tm] V, Vx, IIIx, and more.*
non-EZ	*Devices with CPU MC68328*	*PalmPilot[tm], Palm[tm] III,and more.*
Color	*Devices supporting color*	*Palm[tm] IIIc, and more.*

Palm VII[tm] ROMs

Palm VII**[tm]** ROM Images contain strong encryption technology and by United States law cannot be exported to certain countries. We can attempt to determine the country of origin of your Internet connection. If we determine that you are connecting from approved countries, you will be able to download the files. If we are unable to determine your country of connection (or you are connecting from an embargoed country), you will not be able to get these files from this web site.
Get Palm VII[tm] ROMs.

Palm OS® Software version 3.5

Language	*ROM Type*	*Non-debug*	*Debug*
English	Color	Win / Mac	Win / Mac
English	EZ	Win / Mac	Win / Mac
English	Non-EZ	Win / Mac	Win / Mac
German	Color	Win / Mac	Win / Mac
German	EZ	Win / Mac	Win / Mac

German	Non-EZ	Win / Mac	Win / Mac
Spanish	Color	Win / Mac	Win / Mac
Spanish	EZ	Win / Mac	Win / Mac
Spanish	Non-EZ	Win / Mac	Win / Mac
French	Color	Win / Mac	Win / Mac
French	EZ	Win / Mac	Win / Mac
French	Non-EZ	Win / Mac	Win / Mac
Italian	Color	Win / Mac	
Italian	EZ	Win / Mac	
Italian	Non-EZ	Win / Mac	
Japanese	Color	Win / Mac	
Japanese	EZ	Win / Mac	
Japanese	EZ - 2MB flash	Win / Mac	

There are also pre-release versions of Palm OS software version 3.5 with web clipping wireless support. ROM images with web clipping contain strong encryption technology and by United States law cannot be exported to certain countries. We can attempt to determine the country of origin of your Internet connection. If we determine that you are connecting from approved countries, you will be able to download the files. If we are unable to determine your country of connection (or you are connecting from an embargoed country), you will not be able to get these files from this web site.

Get Web Clipping ROMs.

Palm OS® Software version 3.3

Language	*ROM Type*	*Non-debug*	*Debug*
English	Palm III[tm] organizer	Win / Mac	Win / Mac
English	Palm V[tm], Palm Vx[tm], & Palm IIIx[tm] organizers	Win / Mac	Win / Mac
French	Palm III[tm] organizer	Win / Mac	
French	Palm V[tm], Palm Vx[tm], & Palm IIIx[tm] organizers	Win / Mac	
German	Palm III[tm] organizer	Win / Mac	
German	Palm V[tm], Palm Vx[tm], & Palm IIIx[tm] organizers	Win / Mac	

Palm OS® Software version 3.2

Palm VII**[tm]** ROM images contain strong encryption technology and by United States law cannot be exported to certain countries. We can attempt to determine the country of origin of your Internet connection. If we determine that you are connecting from approved countries, you will be able to download the files. If we are unable to determine your country of connection (or you are connecting from an embargoed country), you will not be able to get these files from this web site.

Get Palm VII[tm] ROMs.

Palm OS® Software version 3.1

Language	*ROM Type*	*Non-debug*	*Debug*
English	Palm V[tm] organizer	Win / Mac	Win / Mac
English	Palm IIIx[tm] & Palm IIIe[tm] organizers	Win / Mac	Win / Mac
French	Palm V & Palm IIIx organizers	Win / Mac	
German	Palm V & Palm IIIx organizers	Win / Mac	
Spanish	Palm V & Palm IIIx organizers	Win / Mac	

Palm OS® Software version 3.0

Language	*ROM Type*	*Non-debug*	*Debug*
English	Palm III[tm] organizer	Win / Mac	Win / Mac
French	Palm III organizer	Win / Mac	
German	Palm III organizer	Win / Mac	
Spanish	Palm III organizer	Win / Mac	

Palm OS® software version 2.0

Language	*ROM Type*	*Non-debug*	*Debug*
English	PalmPilot Professional[tm] organizer	Win / Mac	Win / Mac
English	PalmPilot Personal[tm] organizer	Win / Mac	

Palm OS® software version 1.0

Language	*ROM Type*	*Non-debug*
English	Pilot 1000[tm] & Pilot 5000[tm] organizers	Win / Mac

Digital DNA from Motorola

Motorola Semiconductor Products

Home Solutions Products Support Contact About Us Site Map

SEARCH

ADVANCED

MC68328: DragonBall[tm] Integrated Microprocessor

Product Catalog

Microprocessors

32 Bit

68K/ColdFire

M683XX

As the portable consumer market grows at full speed, system requirements are becoming more rigorous than ever. Minimum components, small board space, low power consumption, and low system cost are several minimum criteria to a successful product. To address these needs, Motorola designed the MC68328 DragonBall[tm] processor. By providing 3.3V, fully static operation in an efficient package, the MC68328 delivers cost-effective performance to satisfy the extensive requirements of today's portable consumer market.

Page Contents

Product Picture

Related Links

General DragonBall Information

MC68328 Features

- Static 68EC000 Core Processor-Identical to MC68EC000 Microprocessor
  - Full Compatibility With MC68000 And MC68EC000
  - 32-Bit both External and Internal Address Bus capable of addressing 4GB Space
  - 16-Bit On-Chip Data Bus For MC68000 Bus Operations
  - Static Design Allows Processor Clock To Be Stopped Providing Dramatic Power Savings
  - 2.7 MIPS Performance At 16.67-MHz Processor Clock
- External M68000 Bus Interface with Dynamic Bus Sizing for 8-bit and 16-bit Data Ports
- System Integration Module (SIM28), Incorporating Many Functions Typically Relegated to External Array Logic, such as:
  - System Configuration, Programmable Address Mapping
  - Glueless Interface to SRAM, EPROM, FLASH Memory
  - Sixteen Programmable Peripheral Chip Selects With Wait State Generation Logic
  - Interrupt Controller with 13 flexible inputs
  - Programmable Interrupt Vector Response For On-Chip Peripheral Modules
  - Hardware Watchdog Timer
  - Software Watchdog Timer
  - Low-Power Mode Control
  - Up to 78-Bit Individually Programmable Parallel I/O Ports
  - PCMCIA 1.0 Support
- UART
  - Support IrDA Physical Layer Protocol
  - 8 Bytes FIFO on Rx and Tx
- Two Separated Serial Peripheral Interface Ports (Master and Slave)
  - Support For External POCSAG Decoder (Slave)
  - Support for Digitizer from A/D Input or EEPROM (Master)
- Dual Channel 16-Bit General Purpose Counter/timer

  - Multimode Operation, Independent Capture/Compare Registers
  - Automatic Interrupt Generation
  - 240-ns Resolution At 16.67-MHz System Clock
  - Each Timer Has An Input And An Output Pin for Capture and Compare
- Pulse Width Modulation Output For Sound Generation
  - Programmable Frame rate
  - 16 Bit programmable
  - Supports Motor Control
- Real Time Clock
  - 24 Hour Time
  - One Programmable Alarm
- Power Management
  - 5 V or 3.3 V Operation
  - Fully Static HCMOS Technology
  - Programmable Clock Synthesizer for Full Frequency Control
  - Low Power Stop Capabilities
  - Modules Can Be Individually Shut-down
  - Lowest Power Mode Control (Shut Down CPU and Peripherals)
- LCD Control Module
  - Software Programmable Screen Size To Support Single (Non-Split) Monochrome/ STN Panels
  - Capable Of Direct Driving Popular LCD Drivers/Modules From Motorola, Sharp, Hitachi, Toshiba etc.
  - Support Up To 4 Grey Levels
  - Utilize System Memory as Display Memory
- IEEE 1149.1 Boundary Scan Test Access Port (JTAG)
- Operation From DC To 16.67 MHz (Processor Clock)
- Operating Voltages of 3.3V ± 0.3V and 5V ± 0.5V
- Compact 144-Lead Thin Quad Flat Pack (TQFP) Package

[top]

MC68328 Parametrics

Processor Speed (MHz)	**Bus Interface** (Bits)	**Performance_1** (MIPS)	**V Voltage** (V)	**Package**
16	32 addr/16 data	2.7 @ 16MHz	3.3 - 5.0	144 TQFP

[top]

MC68328 Documentation

Application Note

Document ID	**Name**	**Type**	**Format**	**Size**	**Rev**	**Date Last Modified**
MC68EZ328DLFLASH	MC68EZ328 Methods of downloading code or data to flash	Application Note	pdf	30k	1	14-DEC-1996
MC68EZ328DTMF	Generating DTMF with PWM module	Application Note	pdf	241k	1	15-FEB-1999
MC68EZ328KEYPAD	MC68EZ328 Minimum I/O to Matrix Keyboard with DragonBall TM EZ328	Application Note	pdf	89k	1	30-SEP-1998
MC68EZ328PLLVCC	PLLVCC Circuit Design for	Application	pdf	38k	1	17-SEP-1998

	DragonBall TM (MC68328) and DragonBall TM -EZ (MC68EZ328)	Note				
MC68EZ328PWM	MC68EZ328 Audio Generation by DragonBall TM MC68EZ328	Application Note	pdf	79k	1	06-NOV-1998
MC68EZ328SRAM16	MC68EZ328 16bit SRAM Interface	Application Note	pdf	70k	1	12-NOV-1998
AN1767/D	MC68328 and MC68EZ328 DragonBall Power Management	Application Note	pdf	26k	0	30-SEP-1998

Fact Sheets

Document ID	Name	Type	Format	Size	Rev	Date Last Modified
M68000MTS	Memory Test Software for 68K	Fact Sheets	pdf	114k	0	01-JAN-1999

Miscellaneous

Document ID	Name	Type	Format	Size	Rev	Date Last Modified
MC68328P/D	MC68328 and MC68328V Integrated Portable System Processor - DragonBall	Miscellaneous	pdf	26k	1	05-OCT-1995
XC68328PV	MC68328 Chip Errata - Mask 3G58E/OH51K	Miscellaneous	pdf	10k	1.0	18-JUL-1996
MC68EZ328CC	Contrast Circuit for Postive VEE	Miscellaneous	pdf	7k	0	04-NOV-1998
MC68EZ328LPF	EZ328 Demo	Miscellaneous	pdf	16k	2	19-APR-1998
MC68EZ328PCMCIA2	MC68EZ328 PCMCIA Release 2.0 Interface Board for DragonBall Update	Miscellaneous	pdf	180k	1	26-NOV-1998

Reference Manual

Document ID	Name	Type	Format	Size	Rev	Date Last Modified
M68000PM/ER	68K Programmer's Ref. Manual Errata	Reference Manual	txt	0k	-	-

---

Users Guide

Document ID	Name	Type	Format	Size	Rev	Date Last Modified
MC68328UM/D	MC68328 DragonBall Microprocessor User's Manual Preliminary	Users Guide	pdf	547k	1	06-NOV-1997
MC68328UMA1/D	MC68328 User's Manual Addendum - LCD Controller Timing Diagram	Users Guide	pdf	17k	0	23-OCT-1998
MC68328UMA2/D	MC68328 User's Manual Addendum - ID Register	Users Guide	pdf	86k	0	14-JAN-1999
MC68328UME/D	MC68328 Spec Errata for DragonBall MC68EZ328 User's Manual	Users Guide	pdf	18k	0	01-NOV-1997
MC68000UM/D	M68000 8-16-32-Bit Microprocessors User's Manual	Users Guide	pdf	998k	9	31-DEC-1993

MC68000UMAD/AD	M68000 8-16-32-Bit Microprocessor User's Manual Addendum	Users Guide	pdf	37k	1	31-DEC-1994

[top]

MC68328 Tools

Tools

Document ID	Name	Type	Format	Size	Rev	Date Last Modified
M68000SC1	Ackerman Benchmark With a Downloadable C Source Code File		txt	0k	-	20-DEC-1993
M68000SC6	Dhrystone 2.1 Benchmarking Part 1 (C Header File)	Dhrystones	txt	17k	2.1	25-MAY-1988
M68000SC7	Fibonacci Benchmark With Downloadable C Source Code File		txt	1k	-	20-DEC-1993
M68000SC8	Sieve Benchmark With Downloadable C Source Code File		txt	1k	-	20-DEC-1993
M68000SC9	Dhrystone 2.1 Benchmarking Part 1	Dhrystones	txt	6k	2.1	25-MAY-1988
M68000BFP	S-Record to C-Struct or Binary File Program	Software Developers Tools	txt	7k	-	-
M68000ASS/SIM	68000 Assembler/Simulator for MS-DOS	Software Developers Tools	zip	156k	-	-
M68000XASS	M680x0 cross assembler MS-DOS	Software Developers Tools	zip	113k	-	-
M68000UNIX	68K Assembler-Berkeley UNIX	Software Developers Tools	arc	67k	-	-
M68000AS332	AS332.ARC-Freeware	Software Developers Tools	arc	57k	-	-
M68000ASMBLR	68K Assembler v 2.71	Software Developers Tools	lzh	54k	-	-
M68000CPLR	68K Compiler	Software Developers Tools	html	0k	-	-
M68000MON	Monitor for 68K Educational Computer Board	Software Developers Tools	zip	148k	-	-
M68000AMPRT	Amiga Port of Matthew Brandt's CC68K Compiler	Software Developers Tools	html	0k	-	-

[top]

MC68328 Design Tools and Data

ID	Name
MC68328ADS	MC68328 Application Development System
MC68328ADSUM/D	MC68328 Application Development System User's

Manual

MC68328SCHEMATICS MC68328 User’s manual ORCAD[tm] Schematics

[top]

SEARCH

ADVANCED

MC68EZ328: DragonBall EZ Integrated Processor

Product Catalog

Microprocessors

32 Bit

68K/ColdFire

M683XX

Related Links

General DragonBall Information

The MC68EZ328 is the second member of the DragonBall[tm] Series of Integrated Portable System Processors.

Inheriting the display capability of the original DragonBall processor, the MC68EZ328 features a more flexible LCD controller with streamlined list of peripherals placed in a smaller package. This processor mainly targeted for portable consumer products which require less peripherals and a more flexible LCD controller. By providing 3V, fully static operation in an efficient 100 TQFP package, the MC68EZ328 delivers cost-effective performance to satisfy the extensive requirements of today's portable consumer market.

Page Contents

MC68EZ328 Features

- Static 68EC000 Core Processor-Identical to MC68EC000 Microprocessor
  - Full Compatibility with MC68000 and MC68EC000
  - 32-Bit internal address bus
  - 24-Bit external address bus capable of addressing maximum 4 x 16MB blocks with chip selects CSA, CSB and 4 x 4 MB blocks with chip selects CSC, CSD.
  - 16-Bit on-chip data bus for MC68000 bus operations
  - Static design allows processor clock to be stopped to provide power savings
  - 2.7 MIPS Performance at 16.58 MHz processor clock
  - External M68000 Bus interface with selectable bus sizing for 8-bit and 16-bit data ports
- System Integration Module (SIM28-EZ), Incorporating Many Functions Typically Related to External Array Logic, such as:
  - System configuration, programmable address mapping
  - Glueless interface to SRAM, EPROM, FLASH memory
  - 8 programmable chip selects with wait state generation logic
  - 4 programmable interrupt I/O and with keyboard interrupt capability
  - 5 general purpose, programmable edge/level/polarity interrupt IRQ
  - Other programmable I/O, multiplexed with peripheral functions up to 47 parallel I/O
  - Programmable interrupt vector response for on-chip peripheral modules
  - Low-Power mode control
- DRAM Controller
  - Support CAS-before-RAS refresh cycles and self-refresh mode DRAM
  - Support 8 bit / 16 bit port DRAM
  - EDO or Automatic Fast Page Mode for LCDC access
  - Programmable refresh rate
  - Support up to 2 banks of DRAM/EDO DRAM

- ○ Programmable column address size

- UART
  - ○ Support IrDA physical layer protocol up to 115.2kbps
  - ○ 8 Bytes FIFO on Tx and 12 Bytes FIFO on Rx
- Serial Peripheral Interface Port
  - ○ 16 bit programmable SPI to support external peripherals
  - ○ Master mode support
- 16-Bit General Purpose Counter / Timer
  - ○ Automatic interrupt generation
  - ○ 60-ns resolution at 16.58-MHz system clock
  - ○ Timer Input/Output pin
- Real Time Clock / Sampling Timer
  - ○ Separate power supply for the RTC
  - ○ One programmable alarm
  - ○ Capable to count up to 512 days
  - ○ Sampling Timer with selectable frequency (4Hz, 8Hz, 16Hz, 32Hz, 64Hz, 256Hz, 512Hz, 1kHz). Generate interrupt for digitizer sampling, or keyboard debouncing.
- LCD Controller
  - ○ Software programmable screen size (up to 640*512) to support single (Non-Split) monochrome/ color STN panels
  - ○ Capable of direct driving popular LCD drivers/modules from Motorola, Sharp, Hitachi, Toshiba etc.
  - ○ Support up to 4 grey levels out of 16 palettes.
  - ○ Utilize system memory as display memory
  - ○ LCD contrast control using 8-bit PWM
- Pulse Width Modulation (PWM) Module
  - ○ 8 bit resolution
  - ○ 5 Byte FIFO provide more flexibility on performance
  - ○ Sound and melody generation
- Build-in Emulation Function
  - ○ Dedicated memory space for Emulator Debug Monitor with Chip Select
  - ○ Dedicated interrupt (Interrupt Level 7) for ICE
  - ○ One address signal comparator and one control signal comparator with masking to support single or multiple Hardware Execution Breakpoint
  - ○ One breakpoint instruction insertion unit
- Boot Strap Mode Function
  - ○ Allow User to initialize system and download program/data to system memory through UART
  - ○ Accept execution command to run program stored in system memory
  - ○ Provide an 8-byte long Instruction Buffer for 68000 instruction storage and execution
- Power Management
  - ○ Fully static HCMOS technology
  - ○ Programmable clock synthesizer using 32.768 kHz/38.4 kHz crystal for full frequency control
  - ○ Low power stop capabilities
  - ○ Modules can be individually shut-down
  - ○ Lowest power mode control
- Operation from DC To 16.58 MHz (processor clock)
- Operating Voltage of 3.0 V to 3.6 V
- Compact 100-Lead Thin Quad Flat Pack (TQFP) and 144 Pin Ball Grid Array (PBGA) packages

[top]

MC68EZ328 Parametrics

Processor Speed (MHz)	Bus Interface (Bits)	Performance_1 (MIPS)	Voltage (V)	Package
20	24 addr/16 data	3.4 @ 20 MHz	3.0 - 3.6	100 TQFP

[top]

MC68EZ328 Documentation

Application Note

Document ID	Name	Type	Format	Size	Rev	Date Last Modified
AN1767/D	MC68328 and MC68EZ328 DragonBall Power Management	Application Note	pdf	26k	0	30-SEP-1998
MC68EZ328DLFLASH	MC68EZ328 Methods of downloading code or data to flash	Application Note	pdf	30k	1	14-DEC-1996
MC68EZ328DTMF	Generating DTMF with PWM module	Application Note	pdf	241k	1	15-FEB-1999
MC68EZ328KEYPAD	MC68EZ328 Minimum I/O to Matrix Keyboard with DragonBall TM EZ328	Application Note	pdf	89k	1	30-SEP-1998
MC68EZ328PWM	MC68EZ328 Audio Generation by DragonBall TM MC68EZ328	Application Note	pdf	79k	1	06-NOV-1998
MC68EZ328SRAM16	MC68EZ328 16bit SRAM Interface	Application Note	pdf	70k	1	12-NOV-1998
MC68EZ328PLLVCC	PLLVCC Circuit Design for DragonBall TM (MC68328) and DragonBall TM -EZ (MC68EZ328)	Application Note	pdf	38k	1	17-SEP-1998

Fact Sheets

Document ID	Name	Type	Format	Size	Rev	Date Last Modified
M68000MTS	Memory Test Software for 68K	Fact Sheets	pdf	114k	0	01-JAN-1999

Miscellaneous

Document ID	Name	Type	Format	Size	Rev	Date Last Modified
MC68EZ328/H	MC68EZ328 DragonBall EZ Integrated Portable System Processor Product Brief	Miscellaneous	pdf	152k	1.3	01-JAN-1998
MC68EZ328CE1J75C	MC68EZ328 - DragonBall-EZ Masket 1J75C	Miscellaneous	pdf	18k	0.1	24-MAR-1998
MC68EZ328CE1J83G	MC68EZ328 - DragonBall-EZ Masket 1J83G	Miscellaneous	pdf	15k	0.1	20-JAN-1999
MC68EZ328PCMCIA2	MC68EZ328 PCMCIA Release 2.0 Interface Board for DragonBall Update	Miscellaneous	pdf	180k	1	26-NOV-1998
MC68EZ328CC	Contrast Circuit for Postive VEE	Miscellaneous	pdf	7k	0	04-NOV-1998

MC68EZ328LPF	EZ328 Demo	Miscellaneous	pdf	16k	2	19-APR-1998

Reference Manual

Document ID	Name	Type	Format	Size	Rev	Date Last Modified
M68000PM/ER	68K Programmer's Ref. Manual Errata	Reference Manual	txt	0k	-	-

Selector Guide

Document ID	Name	Type	Format	Size	Rev	Date Last Modified
SG167/D	68K/Coldfire Product Selector Guide: Networking Systems Division 2nd Q 1999	Selector Guide	pdf	444k	27	31-MAR-1999

Users Guide

Document ID	Name	Type	Format	Size	Rev	Date Last Modified
MC68EZ328UM/D	MC68EZ328 Users Manual - DragonBall-EZ(tm)	Users Guide	pdf	2488k	1	01-NOV-1998
MC68EZ328UMA/D	MC68EZ328 Users Manual Addendum	Users Guide	pdf	16k	1	12-JAN-1999
MC68000UM/D	M68000 8-16-32-Bit Microprocessors User's Manual	Users Guide	pdf	998k	9	31-DEC-1993

[top]

MC68EZ328 Tools

Tools

Document ID	Name	Type	Format	Size	Rev	Date Last Modified
M68000SC1	Ackerman Benchmark With a Downloadable C Source Code File		txt	0k	-	20-DEC-1993
M68000SC6	Dhrystone 2.1 Benchmarking Part 1 (C Header File)	Dhrystones	txt	17k	2.1	25-MAY-1988
M68000SC7	Fibonacci Benchmark With Downloadable C Source Code File		txt	1k	-	20-DEC-1993
M68000SC8	Sieve Benchmark With Downloadable C Source Code File		txt	1k	-	20-DEC-1993
M68000CPLR	68K Compiler	Software Developers Tools	html	0k	-	-
M68000ASS/SIM	68000 Assembler/Simulator for MS-DOS	Software Developers Tools	zip	156k	-	-
M68000AS332	AS332.ARC-Freeware	Software Developers Tools	arc	57k	-	-
M68000UNIX	68K Assembler-Berkeley UNIX	Software Developers	arc	67k	-	-

		Tools				
M68000XASS	M680x0 cross assembler MS-DOS	Software Developers Tools	zip	113k	-	-
M68000MON	Monitor for 68K Educational Computer Board	Software Developers Tools	zip	148k	-	-
M68000ASMBLR	68K Assembler v 2.71	Software Developers Tools	lzh	54k	-	-
M68000AMPRT	Amiga Port of Matthew Brandt's CC68K Compiler	Software Developers Tools	html	0k	-	-
M68000BFP	S-Record to C-Struct or Binary File Program	Software Developers Tools	txt	7k	-	-
M68000SC9	Dhrystone 2.1 Benchmarking Part 1	Dhrystones	txt	6k	2.1	25-MAY-1988

[top]

MC68EZ328 Design Tools and Data

ID	Name
MC68EZ328ADS	MC68EZ328 Application Development System

[top]

Palm.com | MyPalm™ | Wireless | Products | Palm™ Store | Enterprise | Education | Palm OS® | Commun

< Home < Developers < Development Support < Knowledge Base

Developer Knowledge Base

Printer-friendly version Detailed article information

Palm OS®

Home

Palm OS Platform

Palm Economy

Enterprise

Licensing

OEM Partnerships

Alliance Program

Developers

Quick Index

-Start Here-
Conduits
Creator ID
Dev Exchange
Dev Nation

Search

After debugging, my handheld is reset and flashes the "Welcome" s again. What's happening and how do I fix it?

This probably means a breakpoint was left set in the code or that your application itself startup. When the Palm Computing platform device is reset, it sends every application launch code to tell the applications that a reset has occurred. This means your applicati launched after each reset.

There are two common causes for this situation. If the problem is a lingering breakpoin breakpoint is hit (usually the one at PilotMain) and thus the OS tries to break into the d can't because it isn't attached to a debugger any more. The second possibility is that yo doesn't properly handle these reset events from the OS (for example, by attempting to variables that aren't set up), in which case your code itself is the problem and might ca triggering a reboot.

Either way, the device crashes, and the OS resets the device. The process then repeats, Welcome screen flashes up with each reset.

To fix this, hold down the page-up button on the Palm Computing platform device as it until the preferences screen appears). Continuously pressing the page-up button during prevents the device from sending those startup events. You can then delete the offendir and press the reset button once more to return to normal. (If you don't press reset again components of the OS might not be functional since they weren't informed of the reset couldn't initialize themselves.)

Article Information

Article ID:1494
Article Type:FAQ
Article Category: CodeWarrior for Palm Computing Platform

Home | News | Events | Palm OS Platform | Palm Economy | Enterprise
Licensing | OEM Partnerships | Palm Alliance Program | Developers

Terms of Use | Site Map | Contacts

Palm.com | MyPalm™ | Wireless | Products | Palm™ Store | Enterprise | Education | Palm OS® | Commun

< Home < Developers < Development Support < Knowledge Base

Developer Knowledge Base

Palm OS®

Home

Palm OS Platform

Palm Economy

Enterprise

Licensing

OEM Partnerships

Alliance Program

Developers

Quick Index

-Start Here-
Conduits
Creator ID
Dev Exchange
Dev Nation

Search

Printer-friendly version Detailed article information

How can I prevent a Palm OS® platform device from going to sleep a long operation?

EvtResetAutoOffTimer() is the routine that you want to call if you've got a long intens running and need to disable the auto-off feature. The prototype for this function is in S which is inside the "System" folder of your Palm OS Includes folder. You'll have to in SysEvtMgr.h manually; it's not in the default precompiled headers.

When performing a very long operation, you probably want to consider having the ope every so often and call EvtGetEvent(), so that the user can tap a Cancel button, or switc app, or turn off their device. You could either call EvtGetEvent() inside your slow oper dispatch the event as necessary, or you could simply return to your main event loop, ca EvtGetEvent() with a short timeout, and continue the operation when you get a nilEver

Article Information

Article ID:1028
Article Type: FAQ
Article Category: User Interface Manager

Home | News | Events | Palm OS Platform | Palm Economy | Enterprise
Licensing | OEM Partnerships | Palm Alliance Program | Developers

Terms of Use | Site Map | Contacts

Developer Knowledge Base

Palm OS®

Home
Palm OS Platform
Palm Economy
Enterprise
Licensing
OEM Partnerships
Alliance Program
Developers

Quick Index

-Start Here-
Conduits
Creator ID
Dev Exchange
Dev Nation

Go

Search

Printer-friendly version   Detailed article information

How do I programatically add mail to the outbox of the built-in Mail applica

To add mail to the outbox of the built-in mail application, you'll need to populate a MailAddReco information and call SysAppLaunch with the sysAppLaunchCmdAddRecord launch code.

The MailAddRecordParamsType struct definition can be found in AppLaunchCmd.h.

Here is a code example:

```
static void AddToMailApplication()
{
        MailAddRecordParamsPtr theMailInfoPtr;
        LocalID theDBID;
        UInt theCardNo;
        DmSearchStateType theSearchState;
        DWord theResult;

        // Allocate the new pointer for the "add-mail" paramaters
        theMailInfoPtr = MemPtrNew(sizeof(MailAddRecordParamsType));
        if (!theMailInfoPtr)
        return;

        // Setup all the information for our message
        theMailInfoPtr->secret = false;
        theMailInfoPtr->signature = false;
        theMailInfoPtr->confirmRead = false;
        theMailInfoPtr->confirmDelivery = false;
        theMailInfoPtr->priority = mailPriorityNormal;
        theMailInfoPtr->subject = "Subject";
        theMailInfoPtr->from = "a@b.com";
        theMailInfoPtr->to = "b@b.com";
        theMailInfoPtr->cc = "c@b.com";
        theMailInfoPtr->bcc = "d@b.com";
        theMailInfoPtr->replyTo = "e@b.com";
        theMailInfoPtr->body = "Body...";

        // Grab the Local ID and card number of the built-in mail application
        DmGetNextDatabaseByTypeCreator(true, &theSearchState, sysFileTApplication,

        // Finally, tell the mail application to add this item to the outbox
        if (theDBID)
                SysAppLaunch(theCardNo, theDBID, 0, sysAppLaunchCmdAddRecord, (Ptr

        // Finally, free up the memory we allocated for the mail information.
        MemPtrFree( theMailInfoPtr );
}
```

Article Information

Article ID: 1213
Article Type: FAQ

Base=0xFFFFF000

Table 1-3. Programmer's Memory Map

Address	Name	Width	Block	Description	Reset Value(hex)
Base+$000	SCR	8	SIM	System Control Register	$0C
Base+$100	GRPBASEA	16	CS	Chip Select Group A Base Register	$0000
Base+$102	GRPBASEB	16	CS	Chip Select Group B Base Register	$0000
Base+$104	GRPBASEC	16	CS	Chip Select Group C Base Register	$0000
Base+$106	GRPBASED	16	CS	Chip Select Group D Base Register	$0000
Base+$108	GRPMASKA	16	CS	Chip Select Group A Mask Register	$0000
Base+$10A	GRPMASKB	16	CS	Chip Select Group B Mask Register	$0000
Base+$10C	GRPMASKC	16	CS	Chip Select Group C Mask Register	$0000
Base+$10E	GRPMASKD	16	CS	Chip Select Group D Mask Register	$0000
Base+$110	CSA0	32	CS	Group A Chip Select 0 Register	$00010006
Base+$114	CSA1	32	CS	Group A Chip Select 1 Register	$00010006
Base+$118	CSA2	32	CS	Group A Chip Select 2 Register	$00010006
Base+$11C	CSA3	32	CS	Group A Chip Select 3 Register	$00010006
Base+$120	CSB0	32	CS	Group B Chip Select 0 Register	$00010006
Base+$124	CSB1	32	CS	Group B Chip Select 1 Register	$00010006
Base+$128	CSB2	32	CS	Group B Chip Select 2 Register	$00010006
Base+$12C	CSB3	32	CS	Group B Chip Select 3 Register	$00010006
Base+$130	CSC0	32	CS	Group C Chip Select 0 Register	$00010006
Base+$134	CSC1	32	CS	Group C Chip Select 1 Register	$00010006
Base+$138	CSC2	32	CS	Group C Chip Select 2 Register	$00010006
Base+$13C	CSC3	32	CS	Group C Chip Select 3 Register	$00010006
Base+$140	CSD0	32	CS	Group D Chip Select 0 Register	$00010006
Base+$144	CSD1	32	CS	Group D Chip Select 1 Register	$00010006
Base+$148	CSD2	32	CS	Group D Chip Select 2 Register	$00010006
Base+$14C	CSD3	32	CS	Group D Chip Select 3 Register	$00010006
Base+$200	PLLCR	16	PLL	PLL Control Register	$2400
Base+$202	PLLFSR	16	PLL	PLL Frequency Select Register	$0123
Base+$204	Reserved	-	PLL	Do Not Access	-
Base+$207	PCTLR	8	PCTL	Power Control Register	$1F
Base+$300	IVR	8	INTR	Interrupt Vector Register	$00
Base+$302	ICR	16	INTR	Interrupt Control Register	$0000
Base+$304	IMR	32	INTR	Interrupt Mask Register	$00FFFFFF
Base+$308	IWR	32	INTR	Interrupt Wakeup Enable Register	$00FFFFFF
Base+$30C	ISR	32	INTR	Interrupt Status Register	$00000000
Base+$310	IPR	32	INTR	Interrupt Pending Register	-
Base+$400	PADIR	8	PIO	Port A Direction Register	$00
Base+$401	PADATA	8	PIO	Port A Data Register	$00
Base+$403	PASEL	8	PIO	Port A Select Register	$00
Base+$408	PBDIR	8	PIO	Port B Direction Register	$00
Base+$409	PBDATA	8	PIO	Port B Data Register	$00
Base+$40B	PBSEL	8	PIO	Port B Select Register	$00
Base+$410	PCDIR	8	PIO	Port C Direction Register	$00
Base+$411	PCDATA	8	PIO	Port C Data Register	$00
Base+$413	PCSEL	8	PIO	Port C Select Register	$00
Base+$418	PDDIR	8	PIO	Port D Direction Register	$00
Base+$419	PDDATA	8	PIO	Port D Data Register	$00
Base+$41A	PDPUEN	8	PIO	Port D Pullup Enable Register	$FF
Base+$41C	PDPOL	8	PIO	Port D Polarity Register	$00
Base+$41D	PDIRQEN	8	PIO	Port D IRQ Enable Register	$00

Base =
0xfffff000

Table 1-3. Programmer's Memory Map (Continued)

Address	Name	Width	Block	Description	Reset Value(hex)
Base+$41F	PDIRQEDGE	8	PIO	Port D IRQ Edge Register	$00
Base+$420	PEDIR	8	PIO	Port E Direction Register	$00
Base+$421	PEDATA	8	PIO	Port E Data Register	$00
Base+$422	PEPUEN	8	PIO	Port E Pullup Enable Register	$80
Base+$423	PESEL	8	PIO	Port E Select Register	$80
Base+$428	PFDIR	8	PIO	Port F Direction Register	$00
Base+$429	PFDATA	8	PIO	Port F Data Register	$00
Base+$42A	PFPUEN	8	PIO	Port F Pullup Enable Register	$FF
Base+$42B	PFSEL	8	PIO	Port F Select Register	$FF
Base+$430	PGDIR	8	PIO	Port G Direction Register	$00
Base+$431	PGDATA	8	PIO	Port G Data Register	$00
Base+$432	PGPUEN	8	PIO	Port G Pullup Enable Register	$FF
Base+$433	PGSEL	8	PIO	Port G Select Register	$FF
Base+$438	PJDIR	8	PIO	Port J Direction Register	$00
Base+$439	PJDATA	8	PIO	Port J Data Register	$00
Base+$43B	PJSEL	8	PIO	Port J Select Register	$00
Base+$440	PKDIR	8	PIO	Port K Direction Register	$00
Base+$441	PKDATA	8	PIO	Port K Data Register	$00
Base+$442	PKPUEN	8	PIO	Port K Pullup Enable Register	$FF
Base+$443	PKSEL	8	PIO	Port K Select Register	$FF
Base+$448	PMDIR	8	PIO	Port M Direction Register	$00
Base+$449	PMDATA	8	PIO	Port M Data Register	$00
Base+$44A	PMPUEN	8	PIO	Port M Pullup Enable Register	$FF
Base+$44B	PMSEL	8	PIO	Port M Select Register	$FF
Base+$500	PWMC	16	PWM	PWM Control Register	$0000
Base+$502	PWMP	16	PWM	PWM Period Register	$0000
Base+$504	PWMW	16	PWM	PWM Width Register	$0000
Base+$506	PWMCNT	16	PWM	PWM Counter	$0000
Base+$600	TCTL1	16	Timer	Timer Unit 1 Control Register	$0000
Base+$602	TPRER1	16	Timer	Timer Unit 1 Prescalar Register	$0000
Base+$604	TCMP1	16	Timer	Timer Unit 1 Compare Register	$FFFF
Base+$606	TCR1	16	Timer	Timer Unit 1 Capture Register	$0000
Base+$608	TCN1	16	Timer	Timer Unit 1 Counter	$0000
Base+$60A	TSTAT1	16	Timer	Timer Unit 1 Status Register	$0000
Base+$60C	TCTL2	16	Timer	Timer Unit 2 Control Register	$0000
Base+$60E	TPREP2	16	Timer	Timer Unit 2 Prescaler Register	$0000
Base+$610	TCMP2	16	Timer	Timer Unit 2 Compare Register	$FFFF
Base+$612	TCR2	16	Timer	Timer Unit 2 Capture Register	$0000
Base+$614	TCN2	16	Timer	Timer Unit 2 Counter	$0000
Base+$616	TSTAT2	16	Timer	Timer Unit Status Register	$0000
Base+$618	WCR	16	WD	Watchdog Control Register	$0000
Base+$61A	WCR	16	WD	Watchdog Compare Register	$FFFF
Base+$61C	WCN	16	WD	Watchdog Counter	$0000
Base+$700	SPISR	16	SPIS	SPIS Register	$0000
Base+$800	SPIMDATA	16	SPIM	SPIM Data Register	$0000
Base+$802	SPIMCONT	16	SPIM	SPIM Control/Status Register	$0000
Base+$900	USTCNT	16	UART	UART Status/Control Register	$0000
Base+$902	UBAUD	16	UART	UART Baud Control Register	$003F
Base+$904	URX	16	UART	UART RX Register	$0000

Base=
0xfffff000

Table 1-3. Programmer's Memory Map (Continued)

Address	Name	Width	Block	Description	Reset Value(hex)
Base+$906	UTX	16	UART	UART TX Register	$0000
Base+$908	UMISC	16	UART	UART Misc Register	$0000
Base+$A00	LSSA	32	LCDC	Screen Starting Address Register	$00000000
Base+$A05	LVPW	8	LCDC	Virtual Page Width Register	$FF
Base+$A08	LXMAX	16	LCDC	Screen Width Register	$03FF
Base+$A0A	LYMAX	16	LCDC	Screen Height Register	$01FF
Base+$A18	LCXP	16	LCDC	Cursor X Position	$0000
Base+$A1A	LCYP	16	LCDC	Cursor Y Position	$0000
Base+$A1C	LCWCH	16	LCDC	Cursor Width & Height Register	$0101
Base+$A1F	LBLKC	8	LCDC	Blink Control Register	$7F
Base+$A20	LPICF	8	LCDC	Panel Interface Config Register	$00
Base+$A21	LPOLCF	8	LCDC	Polarity Config Register	$00
Base+$A23	LACDRC	8	LCDC	ACD (M) Rate Control Register	$00
Base+$A25	LPXCD	8	LCDC	Pixel Clock Divider Register	$00
Base+$A27	LCKCON	8	LCDC	Clocking Control Register	$40
Base+$A29	LLBAR	8	LCDC	Last Buffer Address Register	$3E
Base+$A2B	LOTCR	8	LCDC	Octet Terminal Count Register	$3F
Base+$A2D	LPOSR	8	LCDC	Panning Offset Register	$00
Base+$A31	LFRCM	8	LCDC	Frame Rate Control Modulation Register	$B9
Base+$A32	LGPMR	16	LCDC	Gray Palette Mapping Register	$1073
					-
Base+$B00	HMSR	32	RTC	RTC Hours Minutes Seconds Register	$00000000
Base+$B04	ALARM	32	RTC	RTC Alarm Register	$00000000
					-
Base+$B0C	CTL	8	RTC	RTC Control Register	$00
Base+$B0E	ISR	8	RTC	RTC Interrupt Status Register	$00
Base+$B10	IENR	8	RTC	RTC Interrupt Enable Register	$00
Base+$B12	STPWCH	8	RTC	Stopwatch Minutes	$00

Note

The base is $FFFFF000 and $FFF000 from reset. If the double-mapped bit is cleared in the SCR, then the base is $FFFFF000. Do not access any space within the 4K register space that is not defined in the above table. Unpredictable results may occur.

6.2 EXCEPTION VECTORS

A vector number is an 8-bit number that can be multiplied by four to obtain the address of an exception vector. An exception vector is the memory location from which the processor fetches the address of a software routine that is used to handle an exception. Each exception has a vector number and an exception vector, as described in Table 6-1. User interrupts are part of the exception processing on the MC68EZ328 and the vector numbers for user interrupts are configurable. For additional information regarding exception processing, see the M68000 Programmer's Reference Manual.

Table 6-1. Exception Vector Assignment

VECTORS NUMBERS		ADDRESS		SPACE	ASSIGNMENT
HEX	DECIMAL	DECIMAL	HEX		
0	0	0	000	SP	Reset: Initial SSP
1	1	4	004	SP	Reset: Initial PC
2	2	8	008	SD	Bus Error
3	3	12	00C	SD	Address Error
4	4	16	010	SD	Illegal Instruction
5	5	20	014	SD	Divide-by-Zero
6	6	24	018	SD	CHK Instruction
7	7	28	01C	SD	TRAPV Instruction
8	8	32	020	SD	Privilege Violation
9	9	36	024	SD	Trace
A	10	40	028	SD	Line 1010 Emulator
B	11	44	02C	SD	Line 1111 Emulator
C	12	48	030	SD	Unassigned, Reserved
D	13	52	034	SD	Unassigned, Reserved
E	14	56	038	SD	Unassigned, Reserved
F	15	60	03C	SD	Uninitialized Interrupt Vector
10–17	16–23	64	040	SD	Unassigned, Reserved
		92	05C		
18	24	96	060	SD	Spurious Interrupt
19	25	100	064	SD	Level 1 Interrupt Autovector
1A	26	104	068	SD	Level 2 Interrupt Autovector
1B	27	108	06C	SD	Level 3 Interrupt Autovector
1C	28	112	070	SD	Level 4 Interrupt Autovector
1D	29	116	074	SD	Level 5 Interrupt Autovector
1E	30	120	078	SD	Level 6 Interrupt Autovector
1F	31	124	07C	SD	Level 7 Interrupt Autovector

Table 6-1. Exception Vector Assignment (Continued)

20–2F	32–47	128	080	SD	TRAP Instruction Vectors
		188	0BC		
30–3F	48–63	192	0C0	SD	Unassigned, Reserved
		255	0FF		
40–FF	64–255	256	100	SD	User Interrupt Vectors
		1020	3FC		

NOTES:

1. Vector numbers 12–14, 16–23, and 48–63 are reserved for future enhancements by Motorola. None of your peripheral devices should be assigned to these numbers.
2. Reset vector 0 requires four words, unlike the other vectors which only require two words, and it is located in the supervisor program space.
3. The spurious interrupt vector is taken when there is a bus error indication during interrupt processing.
4. TRAP #n uses vector number 32+ n (decimal).
5. SP denotes supervisor program space and SD denotes supervisor data space.

Note: The MC68EZ328 does not provide autovector interrupts. At system start-up, you need to program the user interrupt vector so that the processor can handle interrupts properly.

6.3 RESET

The reset exception corresponds to the highest exception level. A reset exception is processed for system initialization and to recover from a catastrophic failure. Any processing that is in progress at the time of the reset is aborted and cannot be recovered. Neither the program counter nor the status register is saved. The processor is forced into the supervisor state. The interrupt priority mask is set at level 7. The address in the first 2 words of the reset exception vector is fetched by the processor as the initial SSP (Supervisor Stack Pointer), and the address in the next two words of the reset exception vector is fetched as the initial program counter.

At start-up or reset, the default chip-select ($\overline{\text{CSA0}}$) is asserted and all other chip-selects are negated. You should use $\overline{\text{CSA0}}$ to decode an EPROM/ROM memory space. In this case, the first two long-words of the EPROM/ROM memory space should be programmed to contain the initial SSP and PC. The initial SSP should point to a RAM space and the initial PC should point to the start-up code within the EPROM/ROM space so that the processor can execute the start-up code to bring up the system.

Note: The MC68EZ328 supports the **reset** instruction. However, the $\overline{\text{RESET}}$ pin will not go low when you issue this instruction because it is an input-only signal.

Hacking the Pilot: Bypassing the Palm OS

Edward Keyes
DaggerWare
mistered@1stresource.com

You've gotten your Pilot, installed the SDK, and you can code a standard app without even launching the Simulator. About now you might be asking yourself: "Is this all this cute little device can do? Isn't there more to life than personal productivity applications?"

This article is for you. The Pilot has a lot of potential that can only be reached by going beyond the Palm OS API routines.

I show you how to hack the Pilot to do grayscale, how to access the sound-generation circuitry directly, and how to install a system patch of your very own. I assume you have a fairly high level of Pilot programming proficiency – we're entering the realm of no error checking and soft resets galore.

Abandon thy Simulator, all ye who enter here. The following hacks do not work correctly on the Pilot Simulator, so stock up on AAA batteries for some on-Pilot debugging. That means you had better make sure your HotSync backups are current, too.

That's Right, I Said Grayscale

It is a little-known fact that the Motorola 68328 Dragonball processor in the Pilot includes on-chip support for simulating grayscale on a black and white LCD panel. The processor does this by cycling pixels on and off with each screen refresh. The average duty cycle of each pixel determines its gray shade. Though some have claimed to see pixel flickering, I think the mode is quite smooth.

In its standard black-and-white mode, the 68328 reads screen memory as a bit-for-bit representation of the screen. The Pilot's display memory is 160 lines of 20 bytes each, arranged in left-to-right and top-to-bottom order. Within each byte, the most significant bit is the left-most pixel.

Just as an aside, if you ever need to hack display memory directly (and you almost certainly will at some point), you can get a pointer to this 3200-byte area with:

```
VoidPtr DisplayMemory;
DisplayMemory=WinGetDisplayWindow()->displayAddr;
```

Note, however, that Palm OS windows are prone to move around in memory somewhat irregularly, so be sure to get this pointer anew at least every event cycle.

In grayscale mode, each pixel requires two bits, so display memory now occupies 6400 bytes, consisting of 160 lines of 40 bytes each, arranged the same way as before, with the left-most pixel in the two most significant bits. To do grayscale, you have to access display memory directly, since there are no routines to do it for you. Furthermore, you have to allocate it yourself, either in the dynamic heap or in the static heaps if you're willing to put up with the write protection.

For the ins and outs of the 68328's LCD controller circuitry, I refer you to the *Dragonball User's Manual,* available from Motorola (part number MC68328UM/AD). I highly recommend every Pilot hacker getting a copy. You can also get a version online in PDF format at http://129.38.233.1/aesop/novu/docs/um/328um.pdf.

Going to Grayscale Mode

In this section I digest a chapter of the *Dragonball User's Manual* and tell you what you really need to do, based on my own experimentation. The one routine you need is:

```
static void GraySwitchDisplayMode
  (short int mode, unsigned char **displayaddr)

// This performs all the register accesses to switch
// between grayscale and black and white modes.
// The displayaddr variable is used in the following way:
// on calling this procedure, load displayaddr with a
// pointer to the address of the new display starting
// address.  The function will return a pointer to the
// old address in the same way.  The calling procedure is
// responsible for saving and then giving back the old
// black and white display starting address.
// (mode=1 is switch to grayscale, mode=0 is switch to
// black and white)
{
  ULong *SSA;
  unsigned char *VPW,*PICF,*FRCM,*LBAR,*CKCON;
  unsigned char *oldstart;

  SSA=(ULong *)0xFFFFFA00;
  VPW=(unsigned char *)0xFFFFFA05;
  PICF=(unsigned char *)0xFFFFFA20;
  FRCM=(unsigned char *)0xFFFFFA31;
  LBAR=(unsigned char *)0xFFFFFA29;
  CKCON=(unsigned char *)0xFFFFFA27;
  if (mode==1) { //switch to grayscale
    // save old display starting address
    oldstart=(unsigned char *)*SSA;
    //switch off LCD update temporarily
    *CKCON=*CKCON & 0x7F;
    //set new display starting address
    *SSA=(ULong)*displayaddr;
    //virtual page width now 40 bytes (160 pixels)
    *VPW=20;
    *PICF=*PICF | 0x01; //switch to grayscale mode
    *LBAR=20; //line buffer now 40 bytes
    //register to control grayscale pixel oscillations
    *FRCM=0xB9;
    //let the LCD get to a new frame (20ms delay)
    SysTaskDelay(2);
    //switch LCD back on in new mode
    *CKCON=*CKCON | 0x80;
    //return original display address for switch back
    *displayaddr=oldstart;
  }
  else if (mode==0) { //switch to black and white
    //save old display starting address
    oldstart=(unsigned char *)*SSA;
    //switch off LCD update temporarily
    *CKCON=*CKCON & 0x7F;
    //set new display starting address
    *SSA=(ULong)*displayaddr;
    //virtual page width now 20 bytes (160 pixels)
    *VPW=10;
    *PICF=*PICF & 0xFE; //switch to black and white mode
    *LBAR=10; // line buffer now 20 bytes
    //let the LCD get to a new frame (20ms delay)
    SysTaskDelay(2);
    //switch LCD back on in new mode
    *CKCON=*CKCON | 0x80;
    //return original display address for switch back
    *displayaddr=oldstart;
  }
}
```

The internal registers for the 68328 processor are located starting at address 0xFFFFF000. To access these directly, all you need to do is set a C pointer to the absolute address you want and use assignment by reference.

There are six important registers for controlling the LCD controller submodule that need to be manipulated in the right ways to successfully switch modes. The SSA (screen starting address) register holds a pointer to the 3200 or 6400-byte display memory. Actually, the 68328 is able to perform hardware pans and scans by manipulating the starting address inside a larger virtual display memory, but that's another topic.

The VPW (virtual page width) register stores the number of words to offset one line from the next in memory. This becomes extremely useful for a virtual display memory larger than the screen, but for now we just need 20 or 40 bytes per line.

The LBAR (last buffer address register) contains the number of words needed for a single display line. In cases where you are hardware scrolling the display by fractions of a byte horizontally (yes, this is possible too), it needs to be a word or two larger, to accommodate pixels hanging off the edge. Here it's the same as our VPW value.

The FRCM (frame rate modulation control) register deals with the innards of the gray scale simulation. If every pixel flashed on and off synchronously, it would be very disconcerting. The values in this register are used to determine how adjacent pixels act. Typically the high and low nibbles should be odd numbers differing by two. The default value 0xB9 seems to work quite well on my Pilot, but the optimum timing might change with the manufacturer of your specific LCD panel.

The PICF (panel interface configuration) register controls the LCD bus bandwidth and the display mode. For our needs, the only important bit is bit 0, which is off for black and white, and on for two-bit grayscale mode.

And finally, the CKCON (clocking control) register is needed because a mode switch does not take effect until the LCD can power down and resync on a new frame. `GraySwitchDisplayMode` contains a 20 ms delay for the LCD panel to rest between mode switches. This delay can be bypassed by extremely clever hacking, but true to form, I found something that worked and then stopped looking.

The Grayscale Palette

The two-bit grayscale mode is actually an indexed color mode. The Pilot can display four simultaneous shades out of a palette of seven (including black and white). This is actually extremely useful, since not everyone's LCD screen will have the same characteristics. Between the palette choice and the contrast knob, you should be able to get four distinct shades.

Here's a bit of code to set the palette. This accesses another register in the LCD controller, and doesn't do anything fancy, but it's an easy way of figuring out the translation from gray intensity values to the actual bits to shove into the GPMR (gray palette mapping) register.

```
static void GraySetPalette
  (short int gray0, short int gray1, short int gray2,
  short int gray3)
// This sets the palette of the 4 active shades of gray
// out of a selection of 7. The four parameters represent
// the shades of the 00, 01, 10, and 11 bit patterns,
// respectively.  The palette of 7 is as follows, in
// terms of darkness density:
// 0=0/16, 1=4/16, 2=5/16, 3=8/16, 4=11/16, 5=12/16,
// 6=16/16, and 7=16/16.  There are two active bit values
// for solid black; either will work.
{
  Word *GPMR;
  Word temp;

  GPMR=(Word *)0xFFFFFA32;
  *GPMR=gray2 + (gray3<<4) + (gray0<<8) + (gray1<<12);
}
```

I am in the process of coding up some rudimentary grayscale library routines. While the full suite isn't in good enough shape for this article, feel free to drop by the DaggerWare Web site at http://www.1stresource.com/~mistered/dagger.htm. By the time you read this, I should have source code posted (and maybe by then someone will have figured out how to do a shared library so everyone will only have to install the code once).

Anyway, the above two routines should give you enough to get started. Just allocate a 6400-byte block in the dynamic heap, call `GraySwitchDisplayMode`, set your palette, and you are free and clear to manipulate the bitmap anyway you want.

I should point out that it may be advantageous to create a routine to copy the black and white display to the grayscale display. That way you can take advantage of many Palm user interface features like form drawing.

A caveat to grayscale mode: the Applications dialog box is invisible, since it draws in the one-bit display memory. Be sure to give your user a way to leave your app in a graceful manner. At the very least, switch back to black and white on an `appStopEvent`.

Direct Sound Control

Theoretically, the Pilot should be able to do voice-quality digitized sound. The 68328 has pulse width modulation (PWM) circuitry that essentially lets you turn a signal line high or low with very good timing.

Unfortunately, Motorola intended U.S. Robotics to put a low-pass filter on the output of that signal line in order to blend the up-down-up-down of the single-bit line into a smoother curve. The filter is supposed to take a local average, turning the duty cycle of a fast pulse into an approximate voltage value to pass onto the piezoelectric speaker.

The Pilot does not have such a filter. According to the Palm tech support guys (cool dudes all, I must say, and thanks for everything), the addition of a low-pass filter turned the output volume into almost nothing. So, until the next generation Pilot arrives, we are stuck with hearing the output of that one signal line. Instead of 16-bit sound like CDs, we have one-bit sound to play with.

Theoretically, you ought to be able to hack even one-bit sound into some sort of polyphonic music, or even rudimentary sampled sound. I've heard tales of similar hacks being done on some of the early Apple machines, but that is really beyond the scope of this article. I'm just going to tell you about the registers and let you play around with getting four-voice harmony yourself.

The Sound Registers

The basic structure of the PWM is simply two countdown timers, a width register and a period register. At each clock pulse, both timers are decremented. Starting out, the output signal is high. When the width register hits zero, the output signal goes low. When the period register then hits zero, the signal is reset high again and both registers are reloaded to start the cycle. Loading the width register with half the period value gets you a perfect high-low-high-low square wave output, for example.

Both registers are double-buffered, so you can load new width and period values while the old ones are still counting down. This is very important, because it means that your code doesn't need to be any more tightly timed than one period, which means about a thousand CPU instructions or so. In other words, you can do this from C rather than in assembly.

The PWM circuitry can generate an interrupt on a period compare, but coding Pilot interrupt routines is beyond my expertise. I recommend a synchronous sound generation method, just like the Pilot ROM uses. You just set up a tight `while` loop waiting for a period timeout before loading the new value. Here are the registers you need:

```
Word *PERIOD,*WIDTH,*PWMREG;
PERIOD  = (Word *)0xFFFFF502;
WIDTH   = (Word *)0xFFFFF504;
PWMREG  = (Word *)0xFFFFF500;
```

The PERIOD register contains the number of clock cycles in a period. For audio frequencies, this will typically be on the order of 100 to 1000, depending on your clock divider value.

The WIDTH register contains the number of clock cycles before the output signal changes. Usually you want to calculate your width values as certain fractions of a period, although for free-form sampled sound, both the PERIOD and the WIDTH are freely changeable.

The PWMREG is the pulse width modulator control register. It handles powering up the sound generation circuitry and setting the clock divider. The lowest three bits control the clock divide. Starting from the Pilot's standard 16.67 MHz clock, the input to the PWM registers is divided by a fraction to reduce the effective frequency. The divider codes are: 000 divide by four, 001 divide by eight, 010 divide by 16, 011 divide by 32, 100 divide by 64, 101 divide by 128, 110 divide by 256, and 111 divide by 512. (I should point out that there is an error in the original 68328 manual regarding these values. The correct ones should be obtained from the manual's "addendum and errata" sheet.)

Bit four (0x0010) enables and disables the PWM circuitry (be sure to turn if off to save power when your program ends). Everything else should be set to zero to avoid enabling interrupts and so forth.

Reading bit 15 (0x8000) tells you when you need to load a new period (it's a poor person's interrupt notification). The main part of your sound playing routine is spent in a loop like this:

```
// null loop waiting for new period
while ((*PWMREG & 0x8000)==0) { }
```

This should tell you enough to get started hacking the Pilot's sound circuitry. If you get sampled sound or multipart harmony working, let me know. And definitely keep in mind that there is a hack just waiting to be written to generate the two-tone frequencies that phones use. Just picture a special Graffiti character you could write that would automatically dial a selected phone number in the Address Book.

The Biggie: Trap Patching

Assuming that you figure out how to create two-tone frequencies, how would you actually go about hacking into the Graffiti code in order to activate your routine at the right time? Good question.

You may have given a passing thought or two to how the CodeWarrior linker knows exactly where in ROM all the different Palm OS system routines are so that you can call them in your applications. Well, actually it doesn't. This is the wonder that is the trap dispatcher.

Basically, what happens is that when you want to call a system routine, the compiler generates a certain "illegal" opcode. This causes the CPU to jump to a certain low memory address where a table of pointers to the actual routines are stored. The trap dispatcher looks up the address of the routine you want, and jumps directly there. That way your application won't have to be changed when the Pilot ROMs are updated, since the trap table can just be altered to point to the new location of the routines.

However, what this means for us is that by manipulating the trap table, we can get the Pilot to call our routines instead of the ROM ones. By installing a trap patch onto `SysHandleEvent` for instance, you automatically gain access to the event loop for all running applications, in ROM or in RAM.

Needless to say, this is an immensely powerful tool. It also goes without saying that you can seriously screw up your Pilot this way. In addition, it needs to be said that the Pilot ROMs are, according to the tech support guys, not 100% robust. Sometimes the built-in routines expect things to be a certain way, and if a trap patch screws that up, you're in trouble. That said, basic trap patching is perfectly fine, and main routines like `SysHandleEvent` can withstand a patch with no problem.

Installing Your Patch

Say you want to temporarily install a routine in your application as a trap patch onto `SysHandleEvent`. The appropriate routine to use is `SysSetTrapAddress`. Just give it the address of your patch routine and go. For the trap codes of all the various system routines, take a look in the SYSTRAPS.H header file. Here's some code (but there is something wrong with it):

```
Boolean (*oldtrap) (EventPtr event);

static Boolean MyNewSysHandleEvent (EventPtr event)
{
  Boolean result;

  // do some fun stuff
  result = (*oldtrap) (event);
  return (result);
}

void InstallMyPatch(void)
{
  oldtrap = SysGetTrapAddress (sysTrapSysHandleEvent);
  SysSetTrapAddress
    (sysTrapSysHandleEvent, MyNewSysHandleEvent);
}
```

The installation is exactly what you need to do, but have a look at the part where the trap routine calls the original ROM routine. It accesses a global variable from a trap. You can't do that without some really special assembly language hacking, because the system trap routines get called from all over the place. Chances are the global variable area is set to point to the Palm OS system globals instead of your own application's a lot of that time.

The easiest way to get around that is to use the Pilot Feature Manager to save and restore the original trap result:

```
static Boolean MyNewSysHandleEvent (EventPtr event)
{
  Boolean result;
  Boolean (*oldtrap) (EventPtr event);

  // do some fun stuff
  FtrGet (MyCreatorCode, MyID, &oldtrap);
  result = (*oldtrap)(event);
  return (result);
}

void InstallMyPatch(void)
{
  DWord oldtrap;

  oldtrap = SysGetTrapAddress (sysTrapSysHandleEvent);
  FtrSet (MyCreatorCode,MyID,oldtrap);
  SysSetTrapAddress(
    sysTrapSysHandleEvent, MyNewSysHandleEvent);
}
```

Depending on your compiler preferences, you might need to do some explicit type casting to force procedure pointers into `DWords` and back again, but the above is the basic idea. Note that there are different ways your trap routine can act. It can:

- Replace the system routine altogether;
- Do something special and then call the system routine;
- Call the system routine with altered parameters; or
- Call the system routine and then fiddle with the results.

The possibilities are endless.

The above example is not overly useful, since as soon as your application quits, your routine goes away. The far better method is to find a way to install your routine permanently in the Pilot's RAM, so it can keep patching a trap no matter what program is running.

My guess is that everyone and their sister will be patching SysHandleEvent

Keeping Your Patch In RAM

There are a couple of ways to do this. My original AppHack program allocated a small bit of memory (16 bytes) in the dynamic heap with a construction like this:

```
VoidPtr MyPointer;
MyPointer=MemPtrNew(NumBytes);
MemPtrSetOwner(MyPointer,0);
```

Note that the Palm OS memory management routines automatically clear the dynamic heap of all chunks your program allocates when your application quits. To keep something in there on a semipermanent basis, you need to set the chunk's owner to zero to let the Pilot know that this is a system-owned chunk.

Now that I've told you this, I beg you not to do it. Dynamic heap space on the Pilot is an extremely precious commodity. Unless you need to grab just a few bytes, try something else. Also, the dynamic heap gets wiped with a soft reset, so if you want your patch to survive, you need to put it somewhere else.

Another good method, used by Wes Cherry in his AlarmHack program, is to allocate some space in a static heap and copy your patch routine into the heap. Unfortunately, like the previous method, this requires that you know exactly how long your patch routine is in bytes, making it mostly unsuitable for C patches. Also, it requires some fairly heavy use of the heap management routines, since there is no generic Palm OS procedure to allocate a chunk in a static heap that doesn't belong to a database.

The best method, and the one I recommend, is to compile your trap patch routine into a separate `code` resource in your PRC file, and then just lock that particular resource down in memory. The installation routine looks like this:

```
VoidHand MyHandle;
VoidPtr MyPatch, oldtrap;

MyHandle = DmGetResource ("code', MyPatchID);
MyPatch = MemHandleLock (MyHandle);
oldtrap = SysGetTrapAddress (sysTrapSysHandleEvent);
FtrSet (MyCreatorCode, MyID, oldtrap);
SysSetTrapAddress (sysTrapSysHandleEvent, MyPatch);
// note you do *not* unlock the patch code handle
DmReleaseResource (MyHandle);
```

This method has the possible disadvantage of fragmenting the static heaps, since it forces part of your application's resource file to remain locked in place. Still, unless you're running in low memory conditions, this shouldn't be much of a problem (and there really is no way around it short of accessing the heap structures directly to move your patch to an out-of-the-way place before locking it down).

Linking Patches with MPW

You may have noticed that I kind of glossed over the "compile your trap patch routine in a separate code resource" part above. This is actually a nontrivial thing to do, but it's automatic in MPW once you set up your make file correctly.

You want to link each source code file separately, so you need to add dependency statements to get your patch to compile and link. Here is an excerpt from a make file (a lot of the variables are defined in the example make files that come with the SDK, so this won't run by itself):

```
"{OBJ_DIR}myapp.c.o" ƒ MakeFile "{SRC_DIR}myapp.c"
  {CPP} -o "{OBJ_DIR}myapp.c.o" "{SRC_DIR}myapp.c"
{C_OPTIONS}
"{OBJ_DIR}mypatch.c.o" ƒ MakeFile "{SRC_DIR}mypatch.c"
  {CPP} -o "{OBJ_DIR}mypatch.c.o" "{SRC_DIR}mypatch.c"
{C_OPTIONS}

myapp ƒƒ MakeFile "{OBJ_DIR}myapp.c.o"
"{OBJ_DIR}mypatch.c.o" "{SRC_DIR}myapp.r"
  {LINK} -single -custom -t rsrc -c RSED
"{LIB_DIR}StartupCode.c.o" ∂
    "{OBJ_DIR}myapp.c.o" -o myapp.code
  {LINK} -single -coderesource -rt CODE=1
    -m MyPatchRoutine -t rsrc -c RSED ∂
    "{OBJ_DIR}mypatch.c.o" -o mypatch.code
  {CC} -d RESOURCE_COMPILER {C_OPTIONS} -e
"{SRC_DIR}myapp.r" > myapp.i
  PilotRez -v 1 -t appl -c myAp -it myapp.i -ot "myapp"
  Duplicate -y "myapp" "myapp.prc"
```

The crucial part items are the link options in the second link statement. The `-coderesource` switch causes the linker to compile the patch as a freestanding code resource (also notice the lack of the STARTUP-CODE.C.O file), and the `-m` switch tells the linker that the entry point routine of the code resource should be `MyPatchRoutine`. This allows you to call that routine by jumping to the beginning of the code resource, as we did above. The MYPATCH.C file should have only your `MyPatchRoutine` routine and whatever procedures it calls. The smaller the patch code resource, the less memory you have to lock down.

The resource definition file also needs to be updated to make sure that the patch's code resource gets included in the PRC file. Here is a sample MYAPP.R file:

```
#include <BuildRules.h>
#include <SystemMgr.rh>

include "myapp.code" 'CODE' 1 as sysResTAppCode 1;
include "myapp.code" 'CODE' 0 as sysResTAppCode 0;
include "myapp.code" 'DATA' 0 as sysResTAppGData 0;

include "mypatch.code" 'CODE' 1 as sysResTAppCode My-
PatchID;

include ":Rsc:myapp.rsrc";

resource sysResTAppPrefs 0 {
  30,     // priority
  0x1000,   // stack size
  0x1000    // minHeapSpace
}
```

The three `"myapp.code"` includes import the main application's code and global data resources. The `"mypatch.code"` include statement imports the main code resource of the patch as a `'CODE'` resource with a user-defined resource ID. Since the patch has no global data, the other two includes are not necessary for it. Note that the `-rt CODE=1` statement in the make file above determines the type and ID of the patch code resource in the resource file.

That's all you need to know to code and install your own trap patches. Of course, now we run into the dreaded topic of….

Extension Conflicts

Macintosh users are no doubt familiar with this phenomenon. What happens when two separate patches from two separate programs grab the same trap?

Now that I've told you this, I beg you not to do it.

Well, if they are both well written, they both record the original trap address and jump to it after doing their thing, so the event flow goes from Patch2 to Patch1 to the ROM routine. Note that the last patch installed is the first one called, since it installs itself directly into the trap and acts like Patch1 is the ROM routine.

What happens if Patch1 is now uninstalled? It puts the ROM trap address back into the trap table, and Patch2 is disabled. Or if Patch1 detects that the trap address is no longer its own, it can leave Patch2 functioning, but now Patch2 is jumping to null memory where Patch1 used to be. There really is no way to do this elegantly without some sort of central patch manager.

Luckily, such a central patch manager does exist – a shareware program written by yours truly. It goes by the name of HackMaster, and you can find all the info you can stand at the DaggerWare Web site: http://www.1stresource.com/~mistered/hackmstr.htm.

Basically, what HackMaster lets you do is write a patch without writing an installation app to go along with it, just like an extension on a Mac. HackMaster takes care of installing your patch and maintaining a proper chain of patches that go on the same trap. This allows patches to be installed and removed in any order without a system restart. Plus, it records your currently open suite of patches and gives you the option of automatically reinstalling them after a soft reset (when ordinarily the trap table is reset but your patch is still locked in memory).

If you need to set options for your patch, HackMaster provides a control panel interface. Basically, you link another code resource into your PRC file with an event handler for the form resources you include, and then HackMaster opens your form and passes you events. Global variables are still not allowed, but Feature Manager calls, databases, and dynamic heap allocations are available for you to use.

My guess is that everyone and their sister will be patching `SysHandleEvent`, so definitely give some serious thought as to how you want your patch to interact with others.

Conclusion

Well, that was fun. Now you should have enough info to make your Pilot stand up and do all sorts of neat tricks. Further investigation is needed in all three areas, though, so there's plenty of room for Pilot hackers to make their mark. Can you code up a decent grayscale library? Can you get polyphonic sound from the PWM? Can you figure out how to get access to globals from a system patch? If so, the world of the Pilot hacker is definitely waiting for you. ✔

Pilot Hack Tutorial

By Darrin Massena (darrin@massena.com)
28 Jun 96

This tutorial is meant to show you how to find your way around the Pilot's memory space with the tool Pilot Hack. Once you've downloaded Pilot Hack (pilhack.prc) to your Pilot ("instapp pilhack.prc") and launched it you can begin exploring the contents of that fascinating little black (until now) box.

Pilot Hack displays memory as hex bytes, ASCII or strings and then lets you page up and down to view more. Although it can be fun exploring a 4 gigabyte address space at random, unless you have a lot of spare time on your hands it helps to at least have some kind of idea what is where. I happened to have a lot of time on my hands and here's some of what I found out (all addresses, etc. in this tutorial are in hex unless otherwise specified):

Crude Pilot 1000 Memory Map (not verified on a Pilot 5000 yet but likely to be the same)

```
00000000-000003ff 68k vectors (including traps starting at $80, #15 at $bc)
000000bc          Trap 15 vector (points to $10c03656)
00000000-00007fff RAM ("Dynamic")
00008000-0fffffff faults on access attempt
10000000-10007fff mirror of the 00000000-00007fff range above (RO w/o permission)
10008000-1001ffff more RAM ("Storage") on 128k machine (RO w/o permission)
10020000-1003ffff out of bounds but doesn't cause a fault
10040000-10bfffff faults on access attempt
10c00000-10c7ffff 512K ROM!
10c80000-???????? repeated images of the ROM
fffff000-fffffb12 DragonBall registers
```

Very interesting! The first 32K of the Pilot's address space is RAM that is duplicated (simultaneously mapped to) the range $1000000-$10007fff. The memory from $0000000-$00007fff can be read from, written to, and executed from but the memory starting at $10000000 can only be read or executed -- no writing without gaining permission from an API (I later discovered) called MemSemaphoreReserve. The Pilot appears to support at least a primitive form of hardware memory protection which is used to separate "Dynamic" (Palm's term) from "Storage" RAM and keep rogue programs from trashing persistent system data structures, user data, and other programs accidentally. This is not real data security, just a very sensible precaution that should protect from the majority of random software happenings.

68328 CPU Registers

Using Pilot Hack we can view the 68328 processor's current register state. Just enter "r" into the Graffiti area to get a register dump. Most of these register's contents are random or just a side-effect of Pilot Hack's execution but a few of them are very informative. First, you'll notice that the program counter (PC) is at an address in the Storage range (e.g., $1000dc48). This tells us that program code executes directly from Storage. I won't go into how I uncovered this but another point worth knowing is that the executing code is exactly the code downloaded to the Pilot, not a decompressed or fixed up copy or anything like that.

The second register of note is A7, a.k.a. SP the Stack Pointer, which points to an address in the lower

32K (e.g., $5ea4). Since a program's stack needs to be writeable at all times it must be located in the Dynamic area. In case you're wondering, the standard stack size for a program is 4096 (decimal) bytes, despite the claims in Palm's SDK documentation that the stack is 2K. It may still be wise, however, to assume that only 2K of that stack is 'yours' to use if you plan to be calling any PalmOS APIs since they may consume up to 2K stack themselves.

A third register you should know about is A5, the application data pointer (there's probably a precise official term but I don't know it so this is what I call it). When a Pilot application is launched by the OS, its startup code (the first routine in the code resource) calls the PalmOS API SysAppStartup which allocates space in Dynamic RAM for the application's data, decompresses the application's initialized data into this space, then sets A5 to point to it before returning. From then on, all references to application data are relative to A5 (the MetroWerks compiler makes sure this is the case). Address-relative references can only be +/- 32K which tells us that a Pilot application can (conveniently) have only 64K of application data. Of course, this doesn't include dynamically allocated data, code-relative data (e.g., static strings), resources, or other storage so this shouldn't turn out to be much of a pinch.

Hacking Pilot Hack

You want to hack Pilot Hack? Enter "h <address> <return>" where <address> is the value stored in the PC register and a <return> is the slashing Graffiti stroke from upper-right to lower left. Don't be alarmed when you see the addresses on the left don't include the first two digits of your address -- I needed the screen space so something had to go. You'll see a hex/ASCII dump of Pilot Hack's code (pretty boring, no easter eggs, sorry) followed by (if you scroll down far enough) resources (its icon, etc. -- hard to recognize unless you know what you're looking for) and its initialized data (some strings like "registers", "Write a '?' for some help", etc). This is basically what a Pilot executable PRC file is: a header and a collection of resources, one if which is code, one of which is initialized data (compressed), and an arbitrary number of other resources for icons, menus, forms, strings, bitmaps, whatever.

Ok, so much for what the CPU registers are pointing at. What else is to be found in Pilot memory? From here I'll break memory contents into two categories: hardware stuff and software stuff. Hardware stuff is things that are there because the "DragonBall" 68328 puts them there or requires them to be there, and software stuff is data created by the folks at Palm, primarily stored in the 512K of ROM.

Hardware Stuff

First let's get into the hardware stuff. Many wonderful things can be discovered by consulting the MC68328 DragonBall &trade; User's Manual. Read this cover to digital cover and you will gain endless insights on what your Pilot is capable of. Many of these capabilities are not exposed by the PalmOS so if you want 'em you'll have to be prepared to go direct. *WARNING: one of the primary purposes of the PalmOS (and most operating systems) is to isolate you from the specifics of the hardware. A Pilot application that writes exclusively to the PalmOS APIs has a good chance of being fully portable to other platforms running the PalmOS (e.g., the Emulator running on the Mac, new hardware devices). Writing directly to the hardware is like playing with fire -- you might get burned as USRobotics and their licensees release new hardware, operating system upgrades, etc. Plus, since the multitasking PalmOS manages and uses much of the hardware you may be interested in playing with there's a good chance conflicts will arise with unexpected consequences.* On the other hand, fire is pretty cool and so are some of the nifty DragonBall capabilities the PalmOS doesn't provide access to (yet).

Starting at address $fffff000 you can find the 68328 registers. These are not the CPU registers but a block of registers (~130 total) that control hardware functions like interrupts, parallel I/O (hey, I don't se a parallel port on my Pilot!), audio, timers, serial I/O, the LCD display, and the Real Time Clock. What these registers are do are beyond the scope of this document (and in many cases beyond the scope of my knowledge!) but I'll call out a couple here to give you an idea of what can be found.

Screen Starting Address Register

Early in my hacking of the Pilot I didn't know what any of the APIs were or how to call them but I needed to get some kind of feedback from my test programs as they executed. Text output, serial output, even beeping are all done through APIs I had no clue about. Scrounging through the 68328 User's Manual I found a register called "LSSA", "Screen Starting Address Register". Hmm... very interesting! Its address is (base) $fffff000 + $a00. You can use Pilot Hack to examine this register ("h fffffa00"). I find the value $000063b0 on my Pilot but it may be different on yours. The Pilot's LCD gets its display data directly from this address. Write a byte at $63b0 and it shows up at the upper-left corner of your display. Use Pilot Hack to view screen memory ("h 63b0" or whatever is right for your Pilot) and you'll see (in hex) the values that make up the display, 20 (decimal) bytes per scan line. Since the first scan line of Pilot Hack's display is blank, the first 20 bytes of the dump are zero. Then $80 (the top-left pixel of the 'h'), $46 (part of the '6' and '3'), and so on.

I wrote debugging output directly to the screen until I figured out enough to be able to call the WinDrawChars API ("function Foo returns 17" sure beats "... .. .."). Pilot Hack still reads and writes video memory directly to scroll the screen when in event trace mode. To check this out, enter 'e' to turn event tracing on and drag the pen around on the screen. Enter 'e' again to toggle tracing off. At the time I didn't know about the API WinScrollRectangle but if I were writing Pilot Hack today I'd use WinScrollRectangle for scrolling because it would be safer, more portable, keep my program smaller, and possibly even be faster (if written in assembler or Palm's compiler optimizes better than mine).

I haven't tried this but presumably one could scroll the screen through memory simply by changing the screen starting address. Combined with the other LCD registers for panning this could be a great speedup for games that scroll the screen a lot. Someone should write a GameBoy Emulator for Pilot.

Real Time Clock

This one is kind of fun because it is always changing. Enter "h fffffb00" (be sure to turn off event tracing first, if you haven't already) and the first four hex bytes are the time your Pilot thinks it is in hours (first byte), minutes (second byte), and seconds (third and fourth byte). If you enter "h <return>" you'll force a refresh at the current address and you can see the second and possibly the minute value has incremented. Nice to know that low-level hardware is taking care of this rather than software that can easily get screwed up, say by an interrupt, and lose track of time.

Many of the other 68328 registers look very interesting. Go figure them out and YOU can write the next hacking tutorial.

Software Stuff

On the border between hardware and software are the 68328's Exception Vectors. When an exception of any kind of occurs (e.g., divide by zero, misaligned memory access, reset, interrupt, TRAP) the CPU looks up the appropriate vector in the table starting at memory address $0 and jumps to that vector to handle the exception. When the PalmOS initializes it sets these vectors to point at its handlers for the various exceptions. Sometimes it can be useful to know where these handlers are -- just look them up in the vector table.

TRAPs

The vectors that interested me the most were the TRAP vectors. By disassembling the game Puzzle, I could tell that it accessed operating services by executing a TRAP #15 instruction followed by (what I call) an API code that indicates which API to dispatch to. The Macintosh uses a similar technique. I'm not really a fan of this approach because it introduces a fair amount of overhead on each API call while the trap dispatcher grabs the API code, decodes it, and dispatches to it. The Amiga uses a lower overhead approach which involves an array of API vectors at a known address and indirecting through the vectors to call APIs. Oh well.

I found it difficult to get much of an idea what Puzzle (and other programs I was disassembling) was doing because I didn't know which APIs were being called as a result of their TRAPs. The first step in cracking this nut was to unravel the algorithm the trap dispatcher uses to map from an API code to the API's address. Question is, where is the trap dispatch routine located? Answer: one of the Exception Vectors is used by the CPU to dispatch to the TRAP #15 handler. My 68000 reference manual tells me this vector is at address $bc. Pilot Hack shows that the vector at $bc points to $10c03656 -- the trap 15 dispatcher!

Enter "h 10c03656" into Pilot Hack to dump the code for the trap 15 dispatcher. If you read the ASCII dump down the right-hand side you'll notice, toward the bottom, the string "TrapDispatcher" (well, the 'a' and 'p' are truncated at the screen edge but you get the idea). More later on what this helpful string is doing there. I typed the bytes (instructions) I found at this address into my PC and used a 68000 disassembler to reveal what the code is. The trap dispatcher is a simple routine that uses part of the API code as an index into an API dispatch table full of vectors pointing to the actual APIs.

I added a command to Pilot Hack to perform this API code-to-address translation upon request. Enter "t a000" (or just the "t 0" shorthand) to go to the address of the first API (MemInit).

Function Names

Disassembly of anything, especially something large and complex like the Pilot 512K ROM, can be quite challenging and time consuming without at least *some* sort of guide posts to give you a clue what the code you're disassembling is doing. The gods granted a great gift to hackers when they created the Pilot and it comes in the form of function names. To my amazement when I first started poking around in PRC files I discovered that *every function in a Pilot app is followed by its (length preceeded) name in ASCII*! Just run a program like strings.exe on Giraffe.prc or Puzzle.prc and you will see strings like "__Startup__", "InitGameBoard", "DrawPiece", etc. Further, many of the programs have compiled in Assert-type strings like "Error launching application", "didn't find empty square position", "pos out of bounds".

I have two reactions to this. My first one is "yay! this makes disassembly a piece of cake!" but my second one is "bummer! all these symbols and asserts are wasting my precious Pilot RAM!". Normally we only see this kind of stuff in DEBUG versions of programs. Are all the released Pilot apps DEBUG versions? Are all software developers for Pilot insane? I don't really have the answer to this but I'm hoping Pilot developers will wake up to the kind of code they're generating and start building more efficient programs for their final releases. You may note that Pilot Hack does NOT have any bogus symbols, asserts, or anything of the kind. Sorry if this makes hacking harder but I'm finding my 128K Pilot squeezed enough as it is.

Do the ROMs also have these function names? Take a look. First toggle Pilot Hack into string dump mode ("s <return>"). Now enter "t a1f6". What do you see a few lines down the screen? "WinCreateWindow"!!! So, the ROMs have these symbols too. I don't care as much about wasted ROM space as I do RAM space but this still seems strange (albeit handy for hackers). Perhaps there is a Pilot debugging tool or something that can make use of these function names. Let me know if you have the answer.

Browsing For Strings

With Pilot Hack in string dump mode you can quickly scan through memory for points of interest. To start scanning at the base of the 512K ROM enter "s 10000000". Then just page down until you hit the end or get bored out of your skull, whichever happens first. In addition to the names of all the PalmOS APIs you can find many internal, private routines that don't have TRAP entrypoints. Knowledge of these can be valuable. Some other strings I found amusing:

```
10c2207c "Hackers Rule!!!!" -- I'm not making this up! Totally excellent!
10c4a9e6 "AMX 68000 Kernel ... Copyright (c) 1994-1995 KADAK Products Ltd."
10c601db "EggOfInfiniteWisdom" -- what is this?
10c612c8 The credit strings -- "Brought to you by:", etc.
```

Looks like USRobotics licensed their operating system kernel from another company "KADAK Products Ltd.". You can find them on the Web along with more information about their kernel (sorry, no link handy).

Events

To write Pilot Hack I needed to be able to handle various events like Graffiti handwriting events, button presses, etc. I added an event trace facility to Pilot Hack to view these events and their arguments. Enter "e" (no <return> needed) to turn on event tracing. As you press buttons or move the pen around on the display Pilot Hack will show you a dump of the event your action triggers. Each value in the dump is actually a word although the format of the dump might confuse you into thinking they're bytes. The first word of the dump is the event type ("eType") and you can see some obvious ones (1 = pen down, 2 = pen up, 3 = pen move). The high byte of the second word is a Boolean that indicates whether the pen is down. The third and fourth words are the pen's x and y coordinates. The rest of the values are event specific.

ROMDump

Using a tool like Pilot Hack to view the contents of the Pilot's memory works well for certain jobs (e.g., decoding a trap) but unless you have Pilot versions of all the tools you want to use (yeah, right) eventually you're going to wish you had the Pilot ROMs where the rest of your tools are -- on the PC (or Mac). It didn't take me long to reach this point because my disassembler runs on the PC. So I want the data from the Pilot ROMs on my PC. How do I get it there? One way would be to write a program to send it over the serial line and capture it with a terminal program on the PC side but I didn't yet have any information on the serial APIs. I fumbled around for a bit until I noticed "Giraffe_High_Score.PRC" in the Backup directory of my Pilot Desktop installation. Giraffe has a way to create a file that is automatically transferred to the PC (backed up) when you hotsync. If I could figure out how they did it I could have the Pilot ROMs 'backed up' onto my PC.

Upon examination, Giraffe revealed that it creates a new resource database, sets a special flag on it that indicates that the database should be backed up on hotsync, then creates a new resource in the database into which it writes the high scores. So, all I had to do is add the same code to Pilot Hack but copy ROM data rather than high scores. NOT! High scores are ~100 bytes long, Pilot ROMs are 512K long, my Pilo has 128K of RAM to store the new database in. So, just write the ROMs out in several pieces and stitch them back together on the other side. Given the amount of free memory on my Pilot I imagined each piece would be ~64K in length, so 8 pieces for the whole thing. Not too bad. But when I wrote the code t do this the Hotsync Manager crashed as it was trying to backup my file! Looks like some sort of size limitation. I tried 48K, crash, 32K, crash, then 16K -- it works! 32 hotsyncs and another set of batteries later I have all the data on my PC. Each piece has the PRC header on it so I wrote a little program to strip it off and concatenate all the data into one file.

To do this yourself, use the (undocumented) 'x' command in Pilot Hack. Start by setting the current address to $10c00000 (the base address of the ROMs) "h 10c00000". Writing an "x" causes Pilot Hack to write 16384 (decimal) bytes starting at the current address into a new resource database, then advance the current address by 16384 bytes. At that point you press hotsync to backup the database. It will show up in your Backup directory with the name "ROMDump.PRC". Rename it to "romdump.0". Go back to your Pilot, write "x" again, hotsync again, rename the new file "romdump.1" and do it again 30 more times. This is the true test of how committed you are to hacking the Pilot. Actually, the *truly* committed would find a better way to do this and share it with the rest of us. With modern-day knowledge of the serial API a better solution wouldn't be difficult to write.

Here's the program I used to stitch all those files together:

```
// MakeROM.cpp
// This program reads 32 16384-byte files with the name "romdump" and
// the extension ".n" where n varies from 0 to 31.  The PRC header is
// stripped and the result is written as a single 512K file "Pilot.ROM"

#include <stdio.h>

unsigned char gab[16384];

void main()
{
        FILE *pfilCombo = fopen("Pilot.ROM", "wb");

        for (int i = 0; i < 32; i++) {
                char szFilename[255];
                sprintf(szFilename, "romdump.%d", i);

                FILE *pfil = fopen(szFilename, "rb");
                fseek(pfil, 0x5a, SEEK_SET);
                fread(gab, 16384, 1, pfil);
                fwrite(gab, 16384, 1, pfilCombo);
                fclose(pfil);
        }
        fclose(pfilCombo);
}
```

Conclusion

So now you know your way around the Pilot, have seen a few of the sights, and can start exploring on your own. What will you find? What will you do with what you learn? I'd love to hear about it.

Back to Pilot Software Development

```
                 About - Download - History - Manual - Contributors

PilRC v2.5b4 / PilRCUI                          Aaron Ardiri (aaron@ardiri.com)
User Manual                                                         6 June 2000

Check the PilRC Home Page for the latest version.

Description

     PilRC    A resource compiler for the Palm Computing Platform
     PilRCUI  A form previewer. It launches a window which previews a close
              approximation of forms as they would appear on the Palm.
              Clicking in the content window of PilRCUI will cause it to
              reload the current script.

              NOT UPDATED: - any GUI programmers want to help?

Table of Contents

     Usage
     Understanding the Manual
     Resource Language Reference
     International Support
     Known Bugs
     Further Reading

Usage

     pilrc [-L LANGUAGE] [-I INCLUDE PATH] [-R RESFILE] [-q] file.rcp [output pa
th]
     pilrcui [-L LANGUAGE] file.rcp

     -L LANGUAGE         Generate resource files for target language,
                         LANGUAGE.
     -I INCLUDE PATH     Search INCLUDE PATH when looking for include or
                         bitmap files.
                         NOTE: multiple paths be repeating the -I option.
     -R RESFILE          Output a .res file specifying all the resources
                         emitted by PilRC.
     -H INCFILE          Output a .h file containing auto-generated resource
                         item IDs for resource items that were defined
                         without an ID previously.
                         NOTE: If -H is not specified then undefined IDs are
                         considered errors.
     -q                  Less noisy output, for you minimalists.
     -Fh                 Use Hebrew font widths for AUTO width calculations.
     -Fj                 Use Japanese font widths for AUTO width
                         calculations.
     -F5                 Use Chinese (Big5) font widhts for AUTO width
                         calcuations.
     -Fg                 Use Chinese (GB) font widhts for AUTO width
                         calcuations.
     -Fkm                Use Korean font widhts for AUTO width calcuations
                         (Hanme font)
     -Fkt                Use Korean font widhts for AUTO width calcuations
                         (Hantip font)
     file.rcp            Input file describing the resources to be emitted.
                         Each resource is written as a separate file in the
                         output path directory. The output filename is
                         constructed by appending the hexcode resource ID to
                         the four character resource type.
     output path         Directory where .bin files should be generated.

Example:

     pilrc myprogram.rcp
     pilrc -I c:\resources -L FRENCH myprogram.rcp
     pilrc -I c:\resources -L BIG5 -F5 -R myprogram.res myprogram.rcp c:\output
```

```
Understanding the Manual

Syntax

     Items in all CAPS appear as literals in the file.
     Items enclosed in "<" and ">" are required fields.
     Items enclosed in "[" and "]" are optional fields.

     Each field's required type is indicated by a suffix after the field
     name (see below for types).

Types

     .i    identifier
           example: kFoo
     .c    character
           (may contain normal C style character escapes)
           example: "O"
     .s    string
           (may contain normal C style character escapes)
           example: "Click Me"
     .ss   multi line string
           PilRC will concatenate strings on seperate lines
           example: "Now is the time for all good " \
                            "men to come and aid of their country"
     .n    number
           defined constant or simple arithmetic expression. Valid
           operators are "+", "-", "*" and "/". Precendence is left to
           right, unless changed with the use of parenthesis.
           NOTE: Math calculations are integer based.
           example: 23
                            12+3+1
                            12*(2+3)
                            'PALM'
     .p    position co-ordinate
           may be a number, expression or one of the following keywords.

                AUTO                 Automatic width or height.
                                     Value is computed based on the text in
                                     the item.
                CENTER               Centers the item either horizontally or
                                     vertically.
                CENTER@<coord.n>     Centers the item at the co-ordinate
                                     following.
                RIGHT@<coord.n>      Aligns the item at the right
                                     co-ordinate following.
                BOTTOM@<coord.n>     Aligns the item at the bottom
                                     co-ordinate following.
                PREVLEFT             Previous items left co-ordinate.
                PREVRIGHT            Previous items right co-ordinate.
                PREVTOP              Previous items top co-ordinate.
                PREVBOTTOM           Previous items bottom co-ordinate.
                PREVWIDTH            Previous items width.
                PREVHEIGHT           Previous items height.

           example: PREVRIGHT+2
                            CENTER@80/2

           NOTE: AUTO and CENTER are not valid in arithmetic expressions.

Comments

     Single line comments begin with "//".
     Block comments exist between the "/*" and "*/" tokens.

     NOTE: "//" comments within the definition of objects are treated as
     errors.

Include Files
```

```
     The .rcp file may contain #include directives.
     This allows a programmer to have one header file for their project
     containing pre-defined resource IDs. Source code can reference the
     symbols as can PilRC.

     PilRC understands three include file formats.

          .h            #define <Symbol.i><Value.n>
          .inc          <Symbol.i> equ <Value.n>
          .java, .jav   package <PackageName>

                        public class <ClassName> {
                             public static final short <Symbol.i> =
                        <Value.n>;
                        }

     Once defined, a symbol can be used in place of any number.
     NOTE: #ifdef derivatives are ignored by PilRC.

Resource Language Reference

     The .rcp file may contain the following object definitions:

          FORM                      Form Resource
          MENU                      Form Menu Bar
          ALERT                     Alert Dialog Resource
          VERSION                   Version String
          STRING                    String Resource
          CATEGORIES                Default Category Names
          APPLICATIONICONNAME       Application Icon Name
          APPLICATION               Application Creator Identification
          ICON
          ICONFAMILY                Icon Bitmap Resource
          SMALLICON
          SMALLICONFAMILY           Small Icon Bitmap Resource
          BITMAP
          BITMAPGREY
          BITMAPGREY16              Bitmap Resource
          BITMAPCOLOR
          BITMAPFAMILY
          TRAP                      HackMaster Trap Resource
          FONT                      User Defined Font Resource
          TRANSLATION               Language String Translation Resource

FORM (tFRM)

     FORM ID <FormResourceId.n> AT (<Left.p> <Top.p> <Width.p> <Height.p>)
       [FRAME] [NOFRAME]
       [MODAL]
       [SAVEBEHIND] [NOSAVEBEHIND]
       [USABLE]
       [HELPID <HelpId.n>]
       [DEFAULTBTNID <BtnId.n>]
       [MENUID <MenuId.n>]
     BEGIN
       <OBJECTS>
     END

Where <OBJECTS> is one or more of:

      TITLE                   <Title.s>
      BUTTON                  <Label.s> ID <Id.n> AT ( <Left.p> <Top.p>
                              <Width.p> <Height.p> )
                              [USABLE] [NONUSABLE] [DISABLED] [LEFTANCHOR]
                              [RIGHTANCHOR]
                              [FRAME] [NOFRAME] [BOLDFRAME] [FONT <FontId.n>]
      PUSHBUTTON              <Label.s> ID <Id.n> AT ( <Left.p> <Top.p>
                              <Width.p> <Height.p> )
```

```
                              [USABLE] [NONUSABLE] [DISABLED] [LEFTANCHOR]
                              [RIGHTANCHOR]
                              [FONT <FontId>] [GROUP <GroupId.n>]
      CHECKBOX                <Label.s> ID <Id.n> AT ( <Left.p> <Top.p>
                              <Width.p> <Height.p> )
                              [USABLE] [NONUSABLE] [DISABLED] [LEFTANCHOR]
                              [RIGHTANCHOR]
                              [FONT <FontId.n>] [GROUP <GroupId.n>] [CHECKED]
      POPUPTRIGGER            <Label.s> ID <Id.n> AT ( <Left.p> <Top.p>
                              <Width.p> <Height.p> )
                              [USABLE] [NONUSABLE] [DISABLED] [LEFTANCHOR]
                              [RIGHTANCHOR]
                              [FONT <FontId.n>]
      SELECTORTRIGGER         <Label.s> ID <Id.n> AT ( <Left.p> <Top.p>
                              <Width.p> <Height.p> )
                              [USABLE] [NONUSABLE] [DISABLED] [LEFTANCHOR]
                              [RIGHTANCHOR]
                              [FONT <FontId.n>]
      REPEATBUTTON            <Label.s> ID <Id.n> AT ( <Left.p> <Top.p>
                              <Width.p> <Height.p> )
                              [USABLE] [NONUSABLE] [DISABLED] [LEFTANCHOR]
                              [RIGHTANCHOR]
                              [FRAME] [NOFRAME] [BOLDFRAME] [FONT <FontId.n>]
      LABEL                   <Label.s> ID <Id.n> AT (<Left.p> <Top.p>)
                              [USABLE] [NONUSABLE] [FONT <FontId.n>]
      FIELD                   <Label.s> ID <Id.n> AT ( <Left.p> <Top.p>
                              <Width.p> <Height.p> )
                              [USABLE] [NONUSABLE] [DISABLED] [LEFTALIGN]
                              [RIGHTALIGN]
                              [FONT <FontId.n>] [EDITABLE] [NONEDITABLE]
                              [UNDERLINED]
                              [SINGLELINE] [MULTIPLELINES] [DYNAMICSIZE]
                              [MAXCHARS <MaxChars.n>]
                              [AUTOSHIFT] [NUMERIC] [HASSCROLLBAR]
      POPUPLIST               <Label.s> ID <Id.n> AT ( <Left.p> <Top.p>
                              <Width.p> <Height.p> )
                              [USABLE] [NONUSABLE] [DISABLED] [VISIBLEITEMS
                              <NumVisItems.n>]
                              [FONT <FontId.n>]
      LIST                    <Label.s> ID <Id.n> AT ( <Left.p> <Top.p>
                              <Width.p> <Height.p> )
                              [USABLE] [NONUSABLE] [DISABLED] [VISIBLEITEMS
                              <NumVisItems.n>]
                              [FONT <FontId.n>]
      FORMBITMAP              ~~<Label.s> ID <Id.n>~~ AT (<Left.p> <Top.p>)
                              [BITMAP <BitmapId.n>] [USABLE] [NONUSABLE]
      GADGET                  <Label.s> ID <Id.n> AT ( <Left.p> <Top.p>
                              <Width.p> <Height.p> )
                              [USABLE] [NONUSABLE]
      TABLE                   ~~<Label.s>~~ ID <Id.n> AT ( <Left.p> <Top.p>
                              <Width.p> <Height.p> )
                              [ROWS <NumRows.n>] [COLUMNS <NumCols.n>]
                              [COLUMNWIDTHS <Col1Width.n> ... <ColNWidth.n>]
      SCROLLBAR               <Label.s> ID <Id.n> AT ( <Left.p> <Top.p>
                              <Width.p> <Height.p> )
                              [USABLE] [NONUSABLE] [VALUE <Value.n>] [MIN
                              <MinValue.n>] [PAGESIZE p.n]
                              [MAX <MaxValue.n>] [PAGESIZE <PageSize.n>]
      GRAFITISTATEINDICATOR <Label.s> AT (<Left.p> <Top.p>)

Notes:

   * ID <Id.n> can be replaced with AUTOID. PilRC will assign an identifier
     for each control which specifies AUTOID. This is useful for controls
     which you won't refer to within the application (ie: LABEL's). Auto
     ID's begin at at 9000 and increase sequentially.
   * The bitmap referenced by the FORMBITMAP tag must appear as a seperate
     resource in the rcp file via the BITMAP tag.
   * MAXCHARS is required for FIELD tag to work properly.
   * Any user defined fonts defined by FONT >= 128 and FONT <= 255 must be
     before above the FORM definition using the FONT tag.
   * Any translations defined in the must be declared before the FORM
     definition using the TRANSLATION tag.

Example:

FORM ID 1 AT (2 2 156 156)
  USABLE MODAL
  HELPID 1
  MENUID 1
BEGIN
  TITLE "AlarmHack"
  LABEL "Repeat Datebook alarm sound" AUTOID) AT (CENTER 16)
  PUSHBUTTON "1" ID 2001 AT (20 PrevBottom+2 12) AUTO GROUP 1
  PUSHBUTTON "2" ID 2002 AT (PrevRight+1 PrevTop PrevWidth PrevHeight) GROUP 1
  PUSHBUTTON "3" ID 2003 AT (PrevRight+1 PrevTop PrevWidth PrevHeight) GROUP 1

  LABEL "times. Ring again every" AUTOID AT (CENTER PrevBottom+2) FONT 0

  PUSHBUTTON "never" ID 3000 AT (13 PrevBottom+2 32 12) GROUP 2
  PUSHBUTTON "10 sec" ID 3001 AT (PrevRight+1 PrevTop PrevWidth PrevHeight) GROU
P 2
  PUSHBUTTON "30 sec" ID 3002 AT (PrevRight+1 PrevTop PrevWidth PrevHeight) GROU
P 2
  PUSHBUTTON "1 min" ID 3003 AT (PrevRight+1 PrevTop PrevWidth PrevHeight) GROUP
 2

  LABEL "Alarm sound:" AUTOID AT (24 PrevBottom+4)
  POPUPTRIGGER "" ID 5000 AT (PrevRight+4 PrevTop 62 AUTO) LEFTANCHOR
  LIST "Standard" "Bleep" ID 6000 AT (PrevLeft PrevTop 52 1) VISIBLEITEMS 2 NONU
SABLE
  POPUPLIST ID 5000 6000

  BUTTON "Test" ID 1202 AT (CENTER 138 AUTO AUTO)
  GRAFFITISTATEINDICATOR AT (100 100)
END

MENU (MBAR)

     MENU ID <MenuResourceId.n>
     BEGIN
       <PULLDOWNS>
     END

Where <PULLDOWNS> is one or more of:

     PULLDOWN <PullDownTitle.s>
     BEGIN
       <MENUITEMS>
     END

Where <MENUITEMS> is one or more of:

     MENUITEM <MenuItem.s> ID <MenuItemId.n> [AccelChar.c]
     MENUITEM SEPARATOR

Example:

     MENU ID 100
     BEGIN
       PULLDOWN "File"
       BEGIN
         MENUITEM "Open..." ID 100 "O"
         MENUITEM SEPARATOR
         MENUITEM "Close..." ID 101 "C"
       END
       PULLDOWN "Options"
       BEGIN
```

```
         MENUITEM "Get Info..." ID 200 "I"
       END
     END

ALERT (tALT)

     ALERT ID <AlertResrouceId.n>
       [HELPID <HelpId.n>]
       [DEFAULTBUTTON <ButtonIdx.n>]
       [INFORMATION] [CONFIRMATION] [WARNING] [ERROR]
     BEGIN
       TITLE <Title.s>
       MESSAGE <Message.ss>
       BUTTONS <Button.s> ... <Button.s>
     END

Notes:

   * The DEFAULTBUTTON tag can be used to specify the button number to
     select if the user switches to another application without pressing any
     button in the alert. The argument is the index of the button, where the
     left-most button is at index '0'.

Example:

ALERT ID 1000
  HELPID 100
  DEFAULTBUTTON 1
  CONFIRMATION
BEGIN
  TITLE "AlarmHack"
  MESSAGE "Continuing will cause you 7 years of bad luck\n" \
          "Are you sure?"
  BUTTONS "Ok" "Cancel"
END

VERSION (tVER)

     VERSION ID <VersionResourceId.n> <Version.s>

Example:

     VERSION ID 1 "1.0 beta"

STRING (tSTR)

     STRING ID <StringResourceId.n> <String.ss>
     STRING ID <StringResourceId.n> FILE <StringFile.s>

Example:

     STRING ID 100 "This is a very long string that shows escape characters \n"
\
                   "as well as continued .ss syntax strings"
     STRING ID 101 FILE "string.txt"

CATEGORIES (tAIS)

     CATEGORIES ID <CategoryResourceId.n> <Category1.s> ... <Category2.s>

Notes:

   * The CATEGORIES tag can be used to specify the default category names
     for the application.
     This resource can then be passed to the "CategoryInitialize()" API
     function function to create the category strings in the application
     info block.

Example:
     CATEGORIES ID 100 "Unfiled" "Business" "Personal"
```

APPLICATIONICONNAME (tAIN)

```
     APPLICATIONICONNAME ID <AINResourceId.n> <ApplicationName.s>
```

Example:

```
     APPLICATIONICONNAME ID 100 "AlarmHack"
```

APPLICATION (APPL)

```
     APPLICATION ID <ApplResourceId.n> <APPL.s>
```

Notes:

   * <APPL.s> must be 4 characters long.
   * <APPL.s> represents the creator id for the application, which can be registered at http://www.palm.com/

Example:

```
     APPLICATION ID 1 "PALM"
```

ICON (tAIB)

```
     ICON <IconFileName.s>
     ICONFAMILY <BitmapFileName.s> * 4 [NOCOLORTABLE] [COLORTABLE]
                [TRANSPARENT r g b] [TRANSPARENTINDEX index]
```

circle
21 across
32x32

Notes:

   * The bitmap must be 32x32, 32x22 or 22x22 in dimension.
   * Compression is not available for ICONS.
   * A tAIB resource with ID 1000 is created.
   * ICON creates a monochrome icon resource.
   * ICONFAMILY creates a multibit (1bpp, 2bpp, 4bpp and 256 color) icon resources.
   * The inclusion of the bitmap color table is not recommended as it slows down system performance. PilRC implements a simple color table matching algorithm to match the bitmaps colortable to the system palette.

Example:

```
     ICON "myicon.bmp"
     ICONFAMILY "icon1bpp.bmp" "icon2bpp.bmp" "icon4bpp.bmp" "icon8bpp.bmp"
     ICONFAMILY "icon1bpp.bmp" "" "" "icon8bpp.bmp" TRANSPARENTINDEX 255
```

SMALLICON (tAIB)

```
     SMALLICON <IconFileName.s>
     SMALLICONFAMILY <BitmapFileName.s> * 4 [NOCOLORTABLE] [COLORTABLE]
                     [TRANSPARENT r g b] [TRANSPARENTINDEX index]
```

Notes:

   * The bitmap must be 15x9 in dimension.
   * Compression is not available for ICONS.
   * A tAIB resource with ID 1001 is created.
   * ICON creates a monochrome icon resource.
   * ICONFAMILY creates a multibit (1bpp, 2bpp, 4bpp and 256 color) icon resources.
   * The inclusion of the bitmap color table is not recommended as it slows down system performance. PilRC implements a simple color table matching algorithm to match the bitmaps colortable to the system palette.

Example:

```
     SMALLICON "mysmicon.bmp"
     SMALLICONFAMILY "smic1bpp.bmp" "smic2bpp.bmp" "smic4bpp.bmp" "smic8bpp.bmp"
     SMALLICONFAMILY "smic1bpp.bmp" "" "" "smic8bpp.bmp" TRANSPARENTINDEX 255
```

BITMAP (Tbmp)

```
     BITMAP       ID <BitmapResourceId.n> <BitmapFileName.s>
                  [NOCOMPRESS] [COMPRESS] [FORCECOMPRESS]
     BITMAPGREY   ID <BitmapResourceId.n> <BitmapFileName.s>
                  [NOCOMPRESS] [COMPRESS] [FORCECOMPRESS]
     BITMAPGREY16 ID <BitmapResourceId.n> <BitmapFileName.s>
                  [NOCOMPRESS] [COMPRESS] [FORCECOMPRESS]
     BITMAPCOLOR  ID <BitmapResourceId.n> <BitmapFileName.s>
                  [NOCOLORTABLE] [COLORTABLE]
                  [TRANSPARENT r g b] [TRANSPARENTINDEX index]
                  [NOCOMPRESS] [COMPRESS] [FORCECOMPRESS]
     BITMAPFAMILY ID <BitmapResourceId.n> <BitmapFileName.s> * 4
                  [NOCOLORTABLE] [COLORTABLE]
                  [TRANSPARENT r g b] [TRANSPARENTINDEX index]
                  [NOCOMPRESS] [COMPRESS] [FORCECOMPRESS]
```

black	0,0,0
dark	96,96,96
light	178,178,178
white	255,255,255

black	#00
dark	#60
light	#B2
white	#ff

Notes:

   * BITMAP creates a monochrome bitmap resource.
   * BITMAPGREY creates a 2bpp (4 color grayscale) bitmap resource.
   * BITMAPGREY16 creates a 4bpp (16 color grayscale) bitmap resource.
   * BITMAPCOLOR creates a 256 color bitmap resource.
   * BITMAPFAMILY creates a multibit (1bpp, 2bpp, 4bpp and 256 color) icon resources.
   * The inclusion of the bitmap color table is not recommended as it slows down system performance. PilRC implements a simple color table matching algorithm to match the bitmaps colortable to the system palette.

Example:

```
     BITMAP       ID 1 "mybitmap.bmp" COMPRESS
     BITMAPGREY   ID 2 "bitm2bpp.bmp" COMPRESS
     BITMAPGREY16 ID 3 "bitm4bpp.bmp" COMPRESS
     BITMAPCOLOR  ID 4 "bitm8bpp.bmp" COMPRESS COLORTABLE
     BITMAPFAMILY ID 5 "bitm1bpp.bmp" "bitm2bpp.bmp" "bitm4bpp.bmp" "bitm8bpp.bm
p" COMPRESS
     BITMAPFAMILY ID 6 "bitm1bpp.bmp" "" "" "bitm8bpp.bmp" COMPRESS
     BITMAPFAMILY ID 7 "bitm1bpp.bmp" "" "" "bitm8bpp.bmp" COMPRESS TRANSPARENT
255 255 255
     BITMAPFAMILY ID 8 "bitm1bpp.bmp" "" "" "bitm8bpp.bmp" COMPRESS TRANSPARENTI
NDEX 255
```

TRAP (TRAP)

```
     TRAP ID <TrapId.n> <TrapNumber.n>
```

Notes:

   * DaggerWare's HackMaster required.
   * <TrapId.n> must be greater than 1000.

Example:

```
     TRAP ID 1000 367
```

FONT (NFNT)

```
     FONT ID <FontResourceId.n> FONTID <FontId.n> <FontFileName.s>
```

FONT file format

The file containing the font information is plain ASCII. To understand how it works it is first necessary to understand how a font is arranged in memory. A font consists of four main parts, a header, a bitmap image, a bitmap location table, and an offset/width table. The bitmap image and location table are generated for you 100% automatically.

The ASCII file consists of two parts, the header and the font data (glyph objects). A full font file is provided with the PilRC distribution.

The FONT header has the following fields:

```
     fontType     The purpose of this field is unknown.
                  The ROM fonts define this value to be 36864.
     maxWidth     Defined as "maximum character width".
                  If not set it is automatically set to the width of the
                  widest character.
     kernMax      Defined as "negative of maximum kern value".
                  The purpose of this field is unknown.
     nDescent     Defined as "negative of descent".
                  The purpose of this field is unknown, and is not used in
                  the ROM fonts.
     fRectWidth   Defined as "width of font rectangle".
                  If not set it is automatically set to the width of the
                  widest character.
     fRectHeight  Defined as "height of font rectangle".
                  If not set it is automatically set when the first glyph
                  is defined.
                  All characters must be exactly this height.
     ascent       The number of rows that make up the ascending part of the
                  glyphs.
                  Ascent plus descent equals fRectHeight. This value should
                  be set.
     descent      The number of rows that make up the descending part of
                  the glyphs.
                  Descent plus ascent equals fRectHeight. This value should
                  be set.
     leading      The purpose of this field is unknown.
```

Each glyph has a bitmap, offset, and a width associated with it. The width can be overridden, however it is not recommended as it is set automatically.

Notes:

* FONT declarations must be defined before the font is used.
* <FontId.n> must be between 128 and 255.
* Custom FONT declarations can only be used on Palm Operating System 3.0 and later.
* Palm Operating System 2.0 and before

```
        [source.rcp]
        FONT ID 1000 FONTID 128 "font.txt"

        [source.c]
        void *font128;
        font128=MemHandleLock(DmGetResource('NFNT', 1000));
        UICurrentFontPtr = font128;
```

* Palm Operating System 3.0+

```
        [source.rcp]
        FONT ID 1000 FONTID 128 "font.txt"

        [source.c]
        FontPtr font128;
        font128=MemHandleLock(DmGetResource('NFNT', 1000));
        FntDefineFont(128, font128);
```

* FONT support in PilRC is not complete, and has been reverse engineered.

International Support

PilRC supports a limited form of international tokenization. It works

by substituting strings in the resource definitions with replacements specified in a TRANSLATION section. Multiple translation blocks may be specified in a resource script. The active language is specified with the "-L" flag to PilRC.

Positioning of controls is a large problem if absolute values are used. It is recommended you use AUTO, CENTER and PREVRIGHT et al when defining the contents of your forms. Example:

```
        pilrc -L FRENCH myscript.rcp res
```

TRANSLATION

```
     TRANSLATION <Language.s>
     BEGIN
       <STRINGTRANSLATIONS>
     END
```

Where <STRINGTRANSLATIONS> is one or more of:

```
     <Original.s> = <Translated.ss>
```

Notes:

* Declare a short keyword for long strings, and define native and foriegn translations for it.

Example:

```
     TRANSLATION "FRENCH"
     BEGIN
       "Repeat Datebook alarm sound" = "Répétitions Alarme Agenda"
       "Ring again every" = "Rappel tous les"
     END
```

Known Bugs

* LIST
    o DISABLED does not work.
    o VISIBLEITEMS may be required for list objects to show properly.
* FIELD
    o MAXCHARS required for field control to accept characters to work.
    o NUMERIC doesn't work in Palm Operating System prior to 3.0
* FORM
    o Using NOSAVEBEHIND on Palm Operating System prior to 3.0 may cause errors.
* FONT
    o Developed based on reverse-engineering.
      The complete operation of how Palm handles FONT manipulation is unknown.

Further Reading

For Palm Computing Platform reference information, be sure to also visit the following websites:

o http://www.palmos.com/
  http://www.palm.com/devzone
  Provided by Palm Computing.

o http://www.massena.com
  A great source of low level Palm Computing Platform information.
  Provided by David Massena.

o http://www.roadcoaders.com
  Need some examples? This is a great resource to find plenty of source code for applications written on the Palm Computing Platform.

The Pilot Record Database Format

This document is based on information derived from experiences with PalmOS 1.0.6, PalmOS 2.0, CoPilot, POSE, the Macintosh simulator and other utilities that read or create PDB files. Future versions of software may behave differently from what is describe below.

The PDB File Format Basics

The Pilot Database (PDB) File format can be used to transfer files to the PalmPilot. It is possible to install PDB files to create either resource or record databases through this mechanism. This document only reviews the record database capabilities of the PDB File Format.

The PDB file format is also used to store databases from the PalmPilot on the Macintosh/PC. Pilot databases contain an attribute bit called the Backup Bit. Setting this bit indicates that no custom conduit will be backing up the database and that the database should be backed up during the HotSync process. If you create a database on the PalmPilot and set the Backup Bit, you will find a copy of the database in PDB format in the Backup directory on the computer with which the HotSync was performed.

There is a direct correlation between what is described here, and the PalmPilot record database format as described in Developing PalmOS 3.0 Applications. See Part III, Chapter 1 for further details. Specifically, pages 37 to 41 of that document (pages 34 to 38 of the version for PalmOS 2.0) will give you an overview of the PalmPilot database structure and help you understand the meaning of some of the record attributes and database header fields.

No mnemonics are provided for named constants in this document. If you are using "C" you may wish to review the header file DataMgr.h for public constants for bit flags. Constants are shown using the "C" language convention for hexadecimal numbers (e.g., 0x0A), followed by decimal values where they are not obvious.

Major Sections of the PDB File

The PDB file (from this point forward that term refers to the record database format exclusively) is made up of the following sections: Header, Record List and Data. They are each described in the following paragraphs.

The Header supplies the basics of the file format: file name (on the PalmPilot), various time stamps, version numbers, file attributes, creator and type information, etc.

The Record List enumerates all the records of the file, their attributes and locations within the PDB file.

The Data portion of the PDB file contains the actual AppInfoArea, SortInfoArea and data records. The AppInfoArea and the SortInfoArea are application-specific areas that are optional elements of the PDB file. A PDB file may contain neither, one or both of these areas. The PDB file need not contain any records for HotSync to install the file, however the unsupported FTP code that comes the Metrowerks development tools does not support the installation of files with zero records. Current Palm-supported tools that read PDB files include the Palm Install Tool (to install via HotSync), the Macintosh-only Simulator (via interactive console commands), and the PalmOS Emulator (POSE).

The Header

The header is made up of the following fields.

Field	**Bytes**	**Type**	**Notes**
Name	32	Null-terminated string	This is the name of the database on the PalmPilot device. It need not match the name of the PDB file in the environment in which it is created.
File Attributes	2	Numeric*	0x0002 Read-Only 0x0004 Dirty AppInfoArea 0x0008 Backup this database (i.e. no conduit exists) 0x0010 (16 decimal) Okay to install newer over existing copy, if present on PalmPilot 0x0020 (32 decimal) Force the PalmPilot to reset after this database is installed 0x0040 (64 decimal) Don't allow copy of file to be beamed to other Pilot.
Version	2	Numeric*	Defined by the application.
Creation Date	4	Numeric*	Expressed as the number of seconds since January 1, 1904. **The database will not install if this value is zero.** (PalmOS 1.0.6)
Modification Date	4	Numeric*	Expressed as the number of seconds since January 1, 1904. **The database will not install if this value is zero.** (PalmOS 1.0.6)

Last Backup Date	4	Numeric*	Expressed as the number of seconds since January 1, 1904. This can be left at zero and the database will install.
Modification Number	4	Numeric*	Set to zero.
AppInfoArea	4	Numeric*	The byte number in the PDB file (counting from zero) at which the AppInfoArea is located. This must be the first entry in the Data portion of the PDB file. If this database does not have an AppInfoArea, set this value to zero. See Note A below.
SortInfoArea	4	Numeric	The byte number in the PDB file (counting from zero) at which the SortInfoArea is located. This must be placed immediately after the AppInfoArea, if one exists, within the Data portion of the PDB file. If this database does not have a SortInfoArea, set this value to zero. ***Do not use this. See Note C below for further details.***
Database Type	4	String	Set this to the desired value. Generally it should match the Database Type used by the corresponding application This is 4 characters long and does not have a terminating null.
Creator ID	4	String	Set this to the desired value. Generally it should match the Creator ID used by the corresponding application. In all cases, you should always register your Creator ID before using it. This is 4 characters long and does not have a terminating null.

Unique ID Seed	4	Numeric*	This is used to generate the Unique ID number of subsequent records. This should be set to zero. See Note B below.
NextRecordList ID	4	Numeric*	Set this to zero. The element is used only in the in-memory representation of a PDB file, but exists in the external version for consistency.
Number of Records	2	Numeric*	This contains the number of records

*Please note that the PalmPilot's processor is a member of the Motorola 68000 family. The processor expect Numeric fields to be arranged with the Most Significant Byte coming first as you move through the file. If you are creating your file on a processor family that does not follow this byte ordering, notably Intel processors, pay attention or you will not have the expected results.

Note A: Because of differences between the behavior of software provided on the Macintosh and PC platforms, and bugs which are present, but different, between those platforms, the following advice is given. **If you are going to have an AppInfoArea, the safest prospect, between the two platforms is to have an AppInfoArea of exactly 512 bytes.** If you need a larger area, it is recommended by Palm's Developer Support that you dedicate a record or resource to that purpose. Macintosh users will experience problems if the AppInfoArea is longer than 512. For Windows-based users, the AppInfoArea will be padded with garbage to be exactly 512 bytes, if it is shorter than that. Windows users should not encounter problems when installing an AppInfoArea longer than 512 bytes.

Note B: Various statements have been made about the UniqueID Seed, but they do not appear to be possible to verify. As of PalmOS 2, UniqueIDs *are not preserved* through the process of backing up and re-installing a database to the Pilot. Changes may come about that will make this work in the expected way, but for now, if you want to count on a known value to uniquely identify a record, the developer should assign that number and store it within the data portion of the record. The UniqueID Seed should be set to zero.

Note C: Backup and downloading of the SortInfoArea is not supported by PalmOS. While it is possible to attach a piece of memory to the SortInfoArea pointer within the PalmPilot, the PDB loading and PDB backup process that occurs when the Backup Bit is set to do not support the SortInfoArea. The best solution is to dedicate a record or resource to the storage of whatever information you might want to keep in the SortInfoArea and avoid using the SortInfoArea.

The Record List

The Record List is made up "n" structures, where "n" represents the number of records in the PDB file. Each structure has the following format.

Field	Bytes	Type	Notes
Record Data Offset	4	Numeric*	The byte number in the PDB file (counting from zero) at which the record is located.
Record Attributes	1	Numeric	0x10 (16 decimal) Secret record bit. 0x20 (32 decimal) Record in use (busy bit). 0x40 (64 decimal) Dirty record bit. 0x80 (128, unsigned decimal) Delete record on next HotSync. The least significant four bits are used to represent the category values.
UniqueID	3	Numeric*	Set this to zero and do not try to second-guess what PalmOS will do with this value. See Note B above.

*See note after first table for Numeric field types.

Assembling the File

To create the PDB file, you must assemble the components in this order.

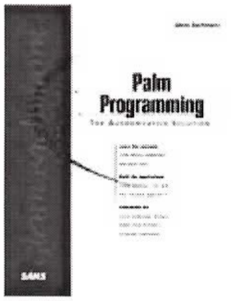
Area	Description
Database Header	The complete header format, as described above.
Record List	Must contain at least one entry.
Filler	Upon transferring a database from the PalmPilot to the desktop environment, the PDB file will have two bytes of filler here. It does not appear to be necessary to insert these two bytes here when creating a PDB file for installation on the PalmPilot. However, if you read a PDB file created by the backup conduit (setting the Backup Bit), you will find that there are two bytes of data in this location.
Data Area	It must be in this order: AppInfoArea (if present), SortInfoArea (if present), and records, sequentially. The order is important because each element's size is computed based on the location of the following element.

Advice

With each release of desktop or device software from Palm, the behavior of the software changes slightly. In an attempt to keep this document up-to-date, please advise the author of any discrepancies or inaccuracies you may observe. Write to bobf@ilx.com or bobf@jhu.edu .

Bachmann Software

Palm Programming: The Authoritative Solution
Chapter 21: Using the Communications Libraries, Part 2: In



Home
Products
Services
Clients
Corporate
News
Jobs
Contact

Note the following is an excerpt from the Macmillan/Sams Publishing book "Palm Programming" by Bachmann Software. You can order the book, which includes the full article as well as a CD-ROM containing all the example code, from Amazon.com

Introduction

In the last chapter, you learned how serial communications can further improve our productivity by providing the opportunity to print and share all of the useful information. Unfortunately, to make it all work, you have to carry along a multitude of cables, wires, adapters, and power supplies. We live in an increasingly wireless world and to maximize the usefulness of our Palm devices (and to send us off the scale in the "cool" department), we need to "beam.". That's right, Star Trek meets Silicon Valley! With infrared communications at our command, we become truly portable (if not the most advanced gadgeteer in the office, which is of course a prize in itself).

There are a few types of infrared (IR) communications available to the Palm user. The specific needs and resources available for a particular application will dictate the approach you take for IR communications on the Palm, although everyone's first desire of to have a deluxe universal remote control is not one of the options. To those of you looking only for this summit of TV enjoyment, I implore you to read on! There are many worthy applications available to the Palm: For instance, take a quick look at your digital cell phone. Many new phones are equipped with built-in IR capability. (Just don't send us the phone bill!).

I continue the discussion of communications topics with the Infrared Library, Palm OS's implementation of the IrDA standard. IrDA (the Infrared Data Association, http://www.irda.org) is the standards organization setting the direction of infrared communications for devices and appliances. Major manufacturers and software houses are members of IrDA today, and this effort will continue to grow in the years ahead. Learning to program to the IrDA specification will put you in the fast lane for Palm OS development. Where else would you want to be? I conclude this chapter by looking at the included sample application, IrDemo. IrDemo provides a framework for writing head-to-head, Palm-to-Palm games (err, I mean applications).

The Standard

The IrDA specifications call for two types of infrared communications, namely SIR (Standard IR) and FIR (Fast IR). SIR effectively provides serial communications replacing the copper wires found in your modem cable with light waves in the infrared spectrum. The transmission speeds of SIR match those of traditional serial communications in the 300[nd] - 115,000 bps range. Fast IR is capable of delivering substantially higher transmission speeds. FIR is being used on some platforms to implement local area networking connectivity because it is capable of throughput of up to 4 Mbits per second. To put this in perspective, your normal Ethernet network is capable of either 10 or 100 Mbits per second.

Like many communications specifications, the IrDA standard defines many protocol layers that form the "IR stack." Each layer offers distinct services upon which additional services are built. At the bottom of the stack is SIR/FIR, which is strictly hardware, involving an IR device that emits light waves and a controller. The controller takes the form of a UART or other chip (for FIR). Riding atop the physical layer is the Infrared Link Access Protocol (IrLAP). This layer provides the actual data path for IrDA communications. There is one IrLAP "connection" per IR device. The Infrared Link Management Protocol (IrLMP) handles one or more sessions over the single IrLAP connection. Higher levels of the stack include: TinyTP, IrCOMM, IrLAN, IrLPT, and OBEX. Additional layers coming into play include protocols for IR keyboards. This is a 10,000- foot view of the stack. You are encouraged to visit the IrDA's Web site for more detailed information.

Not all of the IrDA standards are implemented by Palm OS., However, because the required layers are supported in the IR Library, you can roll your own implementations of the missing layers. The Palm's Exchange Manager implements OBEX, but you will need to develop the code to interact with another devices such as a printer using either IrCOMM or IrLPT. The sample program demonstrates my own IrDemo implementation.

Palm OS IR Capabilities

The first Palm OS release to support infrared communications was Palm OS 3.0. Therefore, IR applications require the 3.0 SDK. The addition in Palm OS 3.0 of primary concern is irlib.h (and the IR Library itself of course). irlib.h lays out all of the data types needed for IR programming. The Palm device's single UART chip provides services for both serial and infrared communications. As I touched upon briefly in the Chapter 20, "Using the Communications Libraries, Part 1: Serial Manager", it is important to reiterate that the Palm III device's sole UART controls both serial and IR communications. This means that you cannot operate the IR port and the serial port simultaneously. The mutual exclusivity between IR and serial communications also has implications for the debugging environment. In short, you cannot use the Metrowerks debugging environment to trace IR calls. In addition, POSE does not currently emulate IR functionality. Included in the IrDemo application is an example of debugging on the Palm, using a home-grown "printf" function.

IR Library Essentials

Before I jump right into the IR Library's API, it is important to have a brief discussion on the architecture of an IR application.

When your Palm application wants to communicate with another device, such as another Palm device, a printer, or cell phone, etc., the first task is to "discover" the other device. This is essentially the IR device looking around for other devices to talk to. Once this process is complete, the application will have the address of the remote device and can initiate a connection. Once you have a low- level connection (IrLAP) to the other device, you need to look up the service you want to communicate with. The Information Access Service (IAS) provides a database of information for the services available for a particular device. When you have received the attribute information from IAS, the application can establish a session with the appropriate service. Let's look at a real world analogy.

Suppose you need an electrician. You open up the Yellow Pages and look under Electricians. This is akin to the IR device performing discovery. It is looking for devices that "speak IR." Once you are on the correct page of the phone book, you review each of the advertisements, selecting the most appealing electrician service and record the phone number. In the same way, a discovery process on the Palm might find many devices, but needs to select one with which to communicate. This will yield an address to use in the connection process.

Okay, so you call the electrician's office and ask for someone with the expertise to install a refrigerator. On the Palm, you initiate an IAS query looking for, say, "IrLPT" for printing services.

The receptionist replies, "Dial extension 123." On the Palm, the IAS query responds with the LSAP selector where IrLPT is found.

So you call extension 123 and speak with the electrician who can help you install the refrigerator. The Palm IR application connects to the IrLPT service and can now print!

The IR Library relies upon two callback functions for event notification. I must provide both of the callback functions in the application, because there is no default event handler provided by Palm OS. The first callback function, named "IrHandler" in IrDemo, shoulders the majority of the workload in the application. IrHandler receives all notifications for IrStack- related events, such as IrLAP connect and disconnect, IrLMP session requests and confirmation, data receipt, discovery completion, and so on. This function is installed during the IrBind function call. The other callback function I have named "IASHandler". IASHandler receives notification when IAS queries have completed.

Because callback events can occur at any time in the Palm application, it is important to avoid the use of alerts or other potentially time-consuming functions. In an effort to provide detailed information during the execution and avoid these problems, IrDemo uses a simple printf function for displaying information. The messages appear in a "window" showing five lines toward the bottom of the Palm's display area.

Without further delay, let's work our way through the IR functions and build our application!

Implementing Infrared Connectivity in a Palm Application

Because it is a shared library (see Chapter 19, "Shared Libraries: Extending the Palm OS"), to use the IR Library, you must first load it with the following code: Err e; // For error result

UInt refNum; // 'Handle' to library

e = SysLibFind(irLibName, &refNum);

(Note: - irLibName is defined in irlib.h.)

After this call, e should contain the value 0, or an error otherwise.

Once the library is loaded, you must open it for the application’s use:

```
e = IrOpen( refNum, irOpenOptSpeed9600 );
```

irOpenOptSpeed9600 is defined in irlib.h along with other constants indicating the initial speed for the port. This is similar to the way in which the Serial Manager works, as described in Chapter 20, "Using the Communications Libraries, Part 1: Serial Manager."

Again, this call should return 0, or an error otherwise.

Once the IrOpen call has been made successfully by an application, it must call IrClose prior to application termination. Due to the way in which Palm applications ’terminate’ when the ’task’ is switched, it is important to detect the application switching to clean up properly.

Now that the library is opened by the application, you need to initialize it with a function named IrBind. Binding will associate an IrConnect structure, defined in the application along with a callback function that the IR Library uses to notify you of completion of certain IR-related events:

```
e = IrBind( refNum, &irCon, IrHandler );
```

See irlib.h for a description of the irCon parameter. It is actually a structure of type IrConnect.

IrBind must return 0, or an error otherwise.

Now that you have bound an IrConnect structure and the IrHandler, you go on to advising the IrStack who we you are. This is done with the IrSetDeviceInfo function. The return value of this function is not the generic Palm OS error type, Err, but rather the type IrStatus. See irlib.h for a description of this type and the possible values it can hold:

```
IrStatus irStat;

static Byte OurDeviceInfo[] = {IR_HINT_PDA, IR_CHAR_ASCII,’P’,’A’,’L’,
’M’,’D’,’E’,’M’,’O’};

if (IrSetDeviceInfo( refNum, OurDeviceInfo ,OurDeviceInfoLen ) !=
IR_STATUS_SUCCESS) {

IrUnbind( refNum, &IrCon );

IrClose( refNum );

printf("IrSetDeviceInfo Failed!" );

return;

}
```

OurDeviceInfo is a byte array, that cannot exceed the size defined in irlib.h of IR_MAX_DEVICE_INFO. This array contains "hint" bytes. The hint bytes are bit masks to indicate the type of device we ’are’ in this application. If there is more than one hint byte to be used, you can use the IR_HINT_EXT, which indicates that the device info contains an additional byte of hint information. Here is an example:

```
// Device info for standard irComm device

static Byte irCommDeviceInfo[] = { IR_HINT_PRINTER|IR_HINT_EXT,
IR_HINT_IRCOMM, IR_CHAR_ASCII, ’I’,’r’,’C’,’O’,’M’,’M’};
```

This function should result in IR_STATUS_SUCCESS.

At this point, you have successfully loaded and opened the IR Library. You have bound it for use and told the IrStack who you are. If you want to advertise a service for other devices to connect to, you make use of the IAS database. This is a ’generic’ database. Each IrDA- compliant device maintains this repository to hold information regarding which services the device offers. This is similar to the way TCP service maps names to ports (ie such as FTP -> 21 or WWW -> 80):

```
static Byte OurDeviceName[] = { IAS_ATTRIB_USER_STRING, IR_CHAR_ASCII,
8,’P’,’A’,’L’,’M’,’D’,’E’,’M’,’O’};

static Byte OurDeviceNameLen = sizeof(OurDeviceName);

/* "Standard" class name for our demo is IrDemo with attribute of
IrDA:IrLMP:LsapSel */

static Byte irdemoQuery[] = { 6,’I’,’r’,’D’,’E’,’M’,’O’,

18,

’I’,’r’,’D’,’A’,’:’,’I’,’r’,’L’,’M’,’P’,

’:’,’L’,’s’,’a’,’p’,’S’,’e’,’l’};

const irdemoQuerySize = sizeof(irdemoQuery);

/* Result for IrDemo */

Byte irdemoResult[] = {

0x01, /* Type for Integer is 1 */

0x00,0x00,0x00,0x02 /* Assumed Lsap */

};

/* IrDemo attribute */

const IrIasAttribute irdemoAttribs = {

(BytePtr) "IrDA:IrLMP:LsapSel",18,

(BytePtr)irdemoResult, sizeof(irdemoResult)};

static IrIasObject irdemoObject ={

(BytePtr)"IrDemo",6,1,

(IrIasAttribute*)&irdemoAttribs};
```

```
IrIAS_SetDeviceName( refNum, OurDeviceName,OurDeviceNameLen);

IrIAS_Add( refNum, &irdemoObject);
```

To connect to another device, you must find the device's address. To obtain this, you use the function IrDiscoverReq:

```
// Initiate a discovery and IrLAP connection

while ( ++lCounter <= lTimeout)

{

irStat = IrDiscoverReq( refNum, &IrCon );

switch (irStat)

{

case IR_STATUS_MEDIA_BUSY:

printf("Media Busy");

continue;

case IR_STATUS_FAILED:

printf("Failed in Discovery");

IrUnbind( refNum, &IrCon );

IrClose( refNum );

FrmCustomAlert(ErrorAlert,

"Failed to Discover. Ending Application","","");

MemSet(&evtExit, sizeof(EventType), 0);

evtExit.eType = appStopEvent;

EvtAddEventToQueue(&evtExit);

return;

case IR_STATUS_PENDING:

// This is the one we want!

// At this point we need to wait for the discovery process to

// complete ...

printf("Discovery Pending!!!");

return;

}
} // while
```

irStat may can come back with one of the following values:

IR_STATUS_MEDIA_BUSY: - indicates that the media is busy and we you should retry the function.

IR_STATUS_FAILED: - indicates an error in the stack.

IR_STATUS_PENDING: - this is the one you want;, it indicates a successful start of the discovery process.

Because it is possible for the IrDiscoverReq function to come back busy a few times and then become 'pending', you wrap this call into a while loop with a timeout on the iterations. This gives the application a healthy chance of finding another device, able to withstand a couple of media busy responses without a disappointing failure in the connection process.

The completion of the discovery process is notified via the callback function registered during the IrBind call, namely IrHandler.

Once discovery has completed successfully, you will have the address for a remote device. Actually, you might have many devices in range, and you will need to sift through them all to select the one you want. You can sort through any available IR devices by examining the hint bytes and "nickname" that is returned by the discovery process. Once you have selected a device, you want to establish an IrLAP connection. You do this with the function IrConnectIrLap:

```
// Check for a valid device

if (pCBParms->deviceList->nItems == 0)

{

printf("No Devices Found!");

return;

}

// At least one device has been found, we will

// assume that the first (and probably only!) device

// found is the one we want.

g_irDevice = pCBParms->deviceList->dev[0].hDevice;

printf("Found %d.%d.%d.%d",g_irDevice.u8[0],g_irDevice.u8[1],

g_irDevice.u8[2],g_irDevice.u8[3]);

// Let's make an IrLAP connection to this address

while ( ++lCounter <= lTimeout)

{
```

```
irStat = IrConnectIrLap( refNum, g_irDevice );
switch (irStat)
{
case IR_STATUS_MEDIA_BUSY:
printf("IrLap Media Busy");
continue;
case IR_STATUS_FAILED:
printf("Failed in IrConnectIrLap" );
return;
case IR_STATUS_PENDING:
// This is the one we want!
// At this point we need to wait for the connect process to
// complete ...
printf("Connect Lap Pending!!!");
return;
}
} // while
```

You are looking for IrLAP to return IR_STATUS_PENDING.

As in the discovery process, you wrap this IrConnectIrLap function in a while loop with a timeout to give it a chance to connect without a single IR_STATUS_MEDIA_BUSY pushing you off course.

When the IrLAP connection has been established, you next want to find out what services the remote device is offering. Here is where the IAS database comes in. You will query the device for a specific service that you are interested in. You will actually execute two IAS queries. The first will demonstrate obtaining the device name. The device name is a required field to be maintained in the IAS, as defined by IrDA:.

```
// Initiate query for remote service we are interested in
// This first query will provide the remote device name
// as defined by the device's IAS
IrIAS_StartResult(&clientQuery);
clientQuery.result = queryResult;
clientQuery.resultBufSize = sizeof(queryResult);
clientQuery.callBack = IASHandler;
clientQuery.queryBuf = irGetQuery;
clientQuery.queryLen = irGetQuerySize;
IrIAS_Query(refNum, &clientQuery);
// Now that we have the device we want
// we need to determine how to connect to it, ie. what lsap?
IrIAS_StartResult(&clientQuery);
clientQuery.result = queryResult;
clientQuery.resultBufSize = sizeof(queryResult);
clientQuery.callBack = IASHandler;
clientQuery.queryBuf = irdemoQuery;
clientQuery.queryLen = irdemoQuerySize;
IrIAS_Query(refNum, &clientQuery);
```

The last area to look at is the IASHandler callback function. This function is invoked when the results of an IAS query are ready. Because the IAS database stores information in an "unstructured" format, each attribute must be stored with a data type identifier. When processing the results of an IAS query, you first look at the data type, and then process the value:

```
switch ( IrIAS_GetType(&clientQuery) )
{
case IAS_ATTRIB_MISSING:
printf("Attribute is Missing?!");
break;
case IAS_ATTRIB_INTEGER:
printf("Get Integer Value");
IrCon.rLsap = IrIAS_GetIntLsap(&clientQuery);
irPack.len = 0;
// We have the address for the service we want to connect to
// Let's establish the LMP session
...
}
```

The connection process "propels" itself along via the callbacks. To review, to connect to a device via the IR Library, the steps are:

1. Load the IR Library using SysLibFind()

2. Open the IR port using IrOpen()

3. Initialize the port using IrBind()

4. Use IrDiscoveryReq() to obtain the device?s address

5. From within the IrHandler callback function, when discovery finishes, we you request the IrLapConnection() with IrLapConnectReq().

6. When this completes, we you query the IAS for the service we you want.

7. When the IAS is complete and IASHandler is called, we you make a request for an LMP connection with IrConnectReq().

8. When this is complete (signalled signaled, of course, by the IrHandler callback), we you have an up- and- running connection to the other device!

IrDemo: Building a Palm OS IR Application

The IrDemo application is designed to be informative and provide you with a launching pad for your own IR projects. You should understand that it is not intended to be a production-ready application. In a number of areas I have left comments for "to- do's", such as handling the case when a connection request is unsuccessful.

All of the code is on the CD-ROM that accompanies this book. To build the application, be sure to compile with the Palm OS 3.0 SDK. To run the application, you will need two Palm devices. The interface has four buttons: Start, Connect, Send, and Finish.

Place the devices head to head so the IR ports can "see" one another. Select Start on both devices. You should see some status information scrolling at the bottom of the display. On one (and only one!) device, select the Connect button. The applications will display status information, indicating the connection activities. Note that the messages will differ on each device. At this point, you will see connection confirmation on both devices, and you may can select the Send button on either device. The data will be received and displayed. When complete, select the Finish button on each device, and the application will terminate.

An interesting thing to try is to move the devices apart and note the messages that are displayed. Move the devices back together (ie, so IR transmission can continue), and notice the display. Move the devices during the discovery process, etc.

The IrDemo application is designed to demonstrate the fundamentals of IrDA. I hope this will help you develop your own applications, whatever they are used for. Our own Bachmann Print Manager product uses infrared connectivity to enable graphics and text printing on popular laser printers. With so many devices supporting the IrDA standard, there is certainly a world of possibilities for creating special capabilities on the Palm device.

For further reading, you can check out the following resources:
-Infrared Data Association (IrDA): http://www.irda.org
-Linux IR Project: - http://www.cs.uit.no/linux-irda/
Chapter 9 of the Palm OS 3.0 documentation

Call Bachmann Software and Services at (973) 729-8628 or e-mail us today!

The IrDA Platform

Stuart Williams and Iain Millar
HP Laboratories, Bristol

Abstract

For almost the past two years the Infrared Data Association has been working to establish an open standard for short range directed Infrared data communications. We are now at a time where the technologies developed within this forum are finding their way into the marketplace. Whilst there has been a high level of multivendor participation and collaboration in the establishing the base level IrDA standards to date very little overview material has been published. This paper provides an introduction to the IrDA's mission and to the technologies that its members have developed. What the IrDA has specified to date is very much a platform. As a platform it meets the key goals of low-cost and multivendor interoperability. It also provides a rich set of ease of use features that will enable multiple applications to concurrently share access to an infrared connection between a pair of devices.

1. Introduction

Since its formation in June 1993 the Infrared Data Association (IrDA) has been working to establish an open standard for short range infrared data communications. At the time of its formation there were a number of vertical, non-interoperable infrared communications technologies. Today IrDA is a strong contender for anyone considering adding infrared data communications to their product. Indeed, whilst supporting their own legacies, vendors who have been offering infrared solutions for years are embarked on the transition to an IrDA based solution.

The key goals for the IrDA are interoperability, low cost, and ease of use. Interoperability is addressed through the creation of an open standard with wide spread, multi-vendor support[1].

Low cost refers to the marginal cost of adding an IrDA interface to products in high volume manufacture. For the most part the cost of adding the digital logic required to provide an IrDA interface is regarded as negligible. The few thousand gates that it takes to implement even the recent higher speed proposals are regarded as coming for free in an environment where ASIC functionality is limited largely by pin-count rather than gate utilisation. This leaves the marginal cost of adding the optoelectronic transceiver which is estimated as $2-$3 and is set to fall further in future with the availability of transceiver modules from optoelectronic suppliers.

Lastly there is ease of use. The IrDA usage model is for short range directed communication link that supports ad-hoc point-and-shoot and place-and-play communications. The nominal operating envelope is a 1m cone with 15 degree half-angles. One of the IrDA frequently asked questions over the past year has been "How do I aim my printer?" The point being that it is all very well to be able to point a PDA at a printer, but it is not really tenable for a printer to sprout legs and point back. Whilst the term "directed" is used to describe an IrDA system, it would be unfair to suggest that it requires highly accurate alignment. Indeed the physical specification allows for more omni-directional behaviour at ranges of less than 1m.

The IrDA system design, which is the focus of the bulk of this paper, is also a significant factor in establishing IrDA platforms as easy to use. Users of conventional communications applications have had to deal with having the correct cables to connect a computer or terminal to a peripheral such as a printer or a modem. They have had to do battle with baud rates, and bits per character and parity. They have also had the responsibility of ensuring that the correct software was loaded at opposite ends of the communications channel.

Whilst the IrDA aims to replace the serial cable for ad-hoc peripheral connection, it also aims to add ease of use features that enable applications to identify peer entities with which they can communicate. Thus a printing subsystem; a file sharing client; a calendar management application; a business card exchange utility... can all identify and locate matching peer entities in order to make use of their services.

The IrDA chose to base its initial standards on a 115kbit/s UART based physical layer that had been developed by Hewlett-Packard (HP-SIR) [1] [2], and an HDLC based Link Access Protocol (IrLAP) originally proposed by IBM [3] [4] [5].

During the course of its first year the need to multiplex multiple application-to-application streams over a single IrLAP connection was identified and with it the need to provide a means for locating and identifying the function of application entities offerings services over an IrDA interface. These needs led to the development of the IrDA Link Management Protocol (IrLMP) [6].

This paper provides an introduction to the services provided by a IrDA platform. These are services upon which new families of Infrared aware applications will be build. End users will not be tied to either applications or platforms from a single vendor.

2. IrDA System Overview

The IrDA Architecture in Figure 2.1.

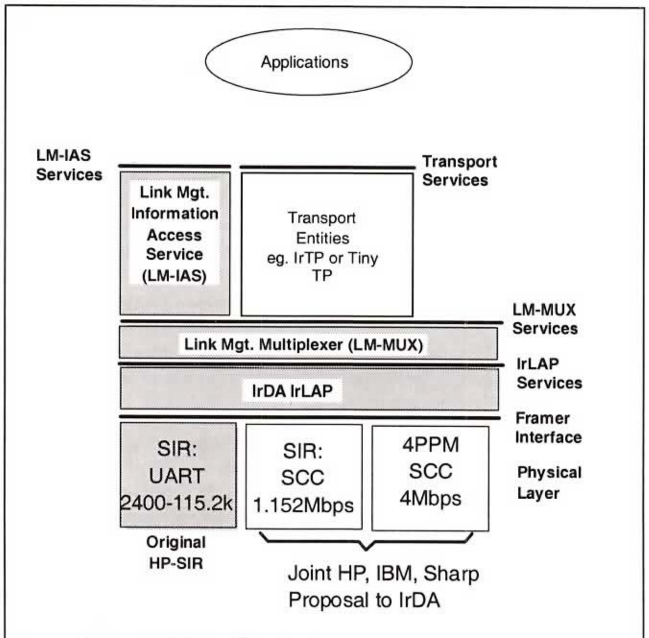


Figure 2.1 IrDA Architecture

There are now three components to the physical IrDA layer[2]:

[1] Including several major manufacturers of computers, PDAs, printers, modem and mobile phones; computer software companies; PTTs and component vendors

[2] The 1.152Mbps and 4Mbps physical layers are currently part of a proposal to the IrDA by HP, IBM and Sharp. It is anticipated that this proposal will be adopted in early 1995.

1. The original 2400bps-115.2kbps HP-SIR [1] based scheme using a conventional UART with character stuffed packet framing

2. A 1.152Mbps scheme that retains the same modulation scheme, but uses a synchronous communications controller and conventional HDLC bit stuffing [7].

3. A 4Mbps scheme that uses a 4PPM modulation scheme and frames packets with a sequence of code violations [7].

From the point of view of the Link Access Protocol (IrLAP) [5], the recent 1.152Mbps and 4Mbps extensions are regarded merely as extra speeds that may be negotiated when a device-to-device connection is established. All three physical layer schemes are designed to have a range of 1m at off axis angle of up to ±15 degrees. In practice, due to component tolerancing, on-axis ranges can be substantially greater, and satisfactory operation can be achieved at off-axis angles of 30 degrees or more.

The Link Access Protocol (IrLAP) is a variation of multi-drop HDLC [3]. It provides facilities for:

1. Controlling Hidden Terminal problems

2. Device Discovery

3. Device-to-device connect/disconnect and QoS negotiation

4. Data Transfer.

IrLAP is an asymmetric protocol and uses HDLC in its normal response mode (NRM). This means that once an IrLAP connection has been established, one station becomes a primary whilst the other becomes a secondary. In the context of a point-to-point connection there is very little difference between the behaviour of primary and secondary stations. However, as we shall see, IrLAP has a the potential to be extended to provide point-to-multipoint device-to-device connectivity. In this case a single primary device would be able to communicate with several secondary devices, but the secondary devices will not be able to communicate directly with each other.

The Link Management Protocol (IrLMP) [6] consists of two parts, a connection oriented multiplexer (LM-MUX) and a directory service (LM-IAS). With the exception of the directory service itself, there are no fixed addresses within the IrDA architecture. Device addresses are chosen at random and exchanged during IrLAP discovery. Address space collisions are resolved by the device that initiates discovery. Likewise 'port' space above LM-MUX is dynamically assigned. The LM-IAS directory service then serves as a means to identify the application services present within a device and the addressing information required establish contact between application peers.

2.1 Addressing

Within the basic IrLAP/IrLMP IrDA platform there are three levels of addressing:

1. Device Addresses: 32-bit randomly chosen identifiers exchanged between devices during IrLAP/IrLMP device discovery.

2. IrLAP Connection addresses: 7 bit HDLC secondary addresses assigned to a secondary device by the primary during IrLAP connection establishment and used for the duration of that connection.

3. IrLMP Multiplexer connection addresses: Logically an LM-MUX service access point is addressed by the concatenation of a 32-bit device address and an 8 bit multiplexer port selector. Once an IrLAP connection is established the IrLAP connection address serves as a synonym for the device address. A multiplexer connection is labelled by the addresses of the LM-MUX service access points at either end of the connection[3]

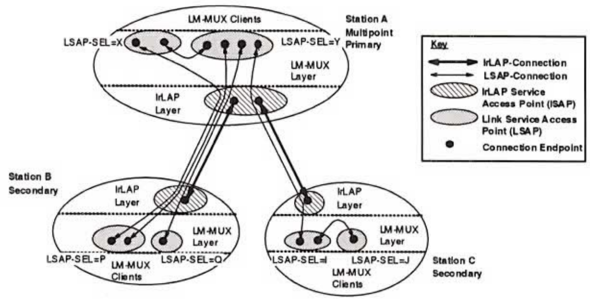


Figure 2.2 IrLAP and IrLMP Connections

The relationship between IrLAP connections, LM-MUX connections/connection end-points and LM-MUX service access points is shown in Figure 2.2

2.2 Link Access Protocol (IrLAP)

IrLAP, the IrDA Link Access Protocol [3], is based on HDLC operating in the Normal Response Mode (NRM) [3]. Typically this mode of operation has been used on multi-drop serial lines between say a terminal controller and a group of terminals sharing the serial line. The terminal controller acts as a primary and regularly polls each of the attached terminals. There are two attractive artefacts to this behaviour in the context of directed short range infrared communications:

1. Once the device-to-device connection has been established then in the absence of aberrant behaviour, access to the shared media is contention free.

2. The constant reversal of the 'line' due to the polling mechanism acts as a beacon to indicate to other devices that approach and active link that the media is in use.

In today's world of peer-to-peer communication a master/slave protocol may seem something of an odd choice. However, it is reasoned that for directed communication the majority of real life scenarios can be addressed by the provision of a single point-to-point link between a pair of devices. In this context the difference between primaries (masters) and secondaries (slaves) becomes moot. Indeed, all differences between primary and secondary are masked by the IrLMP layer above IrLAP so that applications need not be aware of this minor asymmetry.

IrLAP operates in two main modes:

1. Contention Mode: Procedures that occur in contention mode are device discovery, address conflict resolution and connection establishment. All contention mode traffic occurs at 9600bps over the HP-SIR/UART physical layer.

2. Connection mode is entered at the point that an IrLAP connection is established. The communication speed is changed to the rate negotiated in the connection setup messages.

Connection mode traffic has priority over contention mode traffic.

Contention Mode MAC Rules

Once the IrDA stack in a device has been enabled it must sense the media for a minimum of 500ms. Also, a device must sense the media as idle for at least a further 500ms prior to repeating a

[3] This is similar to a TCP/IP connection being labelled by the concatenation of IP address and port number at each end of the connection. Also this leads to the restriction that there may be at most one TCP/IP connection between the same pair of TCP ports. A similar restriction applies to LM-MUX connections.

contention mode procedure. 500ms is absolute upper bound on the time that either a primary or secondary station may retain the right to transmit frames. Shorter intervals may be negotiated during connection establishment.

Connection Mode MAC Rules

Once and IrLAP connection has been established access to the media is mediated by the exchange of a token (the P/F bit in the HDLC control field). Both primary and secondary stations monitor the exchange of this token and provide status indications upward in the event of deteriorating link quality or loss of connection. A station may transmit a number of frames, up to a limit bounded by the negotiated window size, maximum data packet size and the overriding turnaround time for returning the token to its peer.

2.2.1 Hidden Terminal Management

We have already touched on the hidden terminal management capabilities of IrLAP. By placing an absolute upper bound on the link turn around time it is possible to ensure that each end of an IrLAP connection makes regular transmissions that act as a beacon to indicate that there is an established IrLAP connection in the vicinity. Hidden terminals (hidden from one end of the connection) remain silent in the presence of an active IrLAP connection.

2.2.2 Device Discovery and Address Conflict Resolution

An IrDA device address is a 32-bit identifier that a station randomly assigns to itself. Within the relatively small extend of the 'reachspace' the probability of two or more devices choosing the same address is relatively small[4]. IrLAP provides the facility for its client (IrLMP) to instruct devices with colliding addresses to select new device addresses. IrLMP drives the address resolution process by making a single attempt to resolve each address conflict.

Device discovery takes place in contention mode. Device discovery is used to retrieve *<DeviceAddress><DeviceInfo>* tuples from devices in the vicinity.

<DeviceInfo>=<ServiceHints><DeviceNickName>

Service hints is an extensible bit map that provides for a very coarse characterisation of the services offered by the device: currently defined hints bits can specify a device as offering the services of a PDA, a Computer, a Printer, a Modem, a Fax, LAN Access, Telephony, or a File Servers. It is incumbent on the designers of an application service to state what hints bits will be set if an instance of that service is available with a station.

The Device Nickname is a short name that may be presented to the user in order to select between two otherwise identical devices.

A slotted discipline is used for discovery. The station initiating discovery issues a request that specifies the use of 1, 6, 8 or 16 slots. Stations receiving this request randomly select a slot between 0 and the specified upper bound minus 1 in which to make their response. The initiating station then 'calls' out each slot in turn, marking its start with a packet that encodes the slot number slot being polled. Finally the initiating station marks the end of discovery with a final packet that includes the stations own discovery information. Device discovery is illustrated in Figure 2.3.

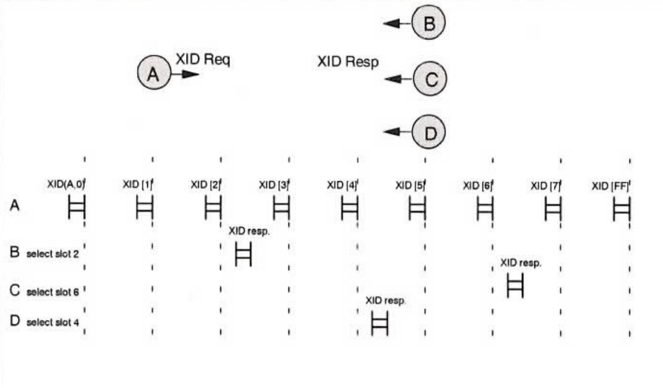


Figure 2.3 Eight Slot Device Discovery

2.2.3 Connection Establishment

An IrLAP connection is initiated by the transmission of a Set Normal Response Mode (SNRM) frame, using Contention mode MAC rules, by the station that will initially become the primary station. The SNRM frame contains a number of negotiable QoS parameters, including: data rate capabilities; turnaround requirements negotiable from 500ms down to 50ms; maximum data packet size; receiver window size, 1 through 7; Link disconnect and threshold times to deal with packet loss. The SNRM also contains a device address (retrieved by a recent discovery operation) and assigns a 7-bit connection address for use during the connection.

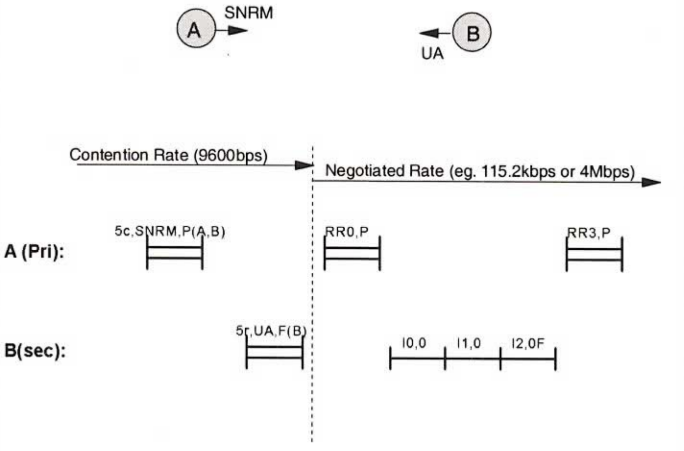


Figure 2.4 IrLAP Connection Establishment

The station addressed by the SNRM responds to the SNRM with an Unnumbered Acknowledge frame, also at the 9600bps contention rate, that contains the results of the negotiation process. At this point both stations apply the newly negotiated parameters. The primary immediately sends an Receiver Ready frame to indicate to the secondary that it is now using the negotiated communication parameters[5].

2.2.4 IrLAP Data Transfer Services

Once an IrLAP connection has been established then the data transfer service between devices is similar to that provided by HDLC operating in Normal Response Mode. It provides for the reliable sequenced exchange of packets. Provision is also made for the transmission of Unnumbered Information (UI) frames. These frames are sent at the negotiated data rate and have

[4] Actually the generation of good quality random numbers is an issue here. If the device address of two devices do collide and they both employ the same pseudo random sequence then their next choice of device address may very well collide as well. Some genuinely random process must be incorporated in the process of assigning a device address.

[5] Careful timing is needed here to ensure that both the primary and secondary have applied the new parameters prior to the transmisson of the RR.

priority over the exchanged of sequence information (I) frames, but they are also subject to loss without recovery.

2.3 Link Management Protocol (IrLMP)

The IrDA Link Management Protocol (IrLMP) [6] provides two distinctly different types of services. Firstly it provides a level of connection oriented multiplexing (LM-MUX) on top of IrLAP. Secondly it provides an Information Base that hold details of the application entities present in the local station that are current offer services to other IrDA devices. Objects in this information base carry the essential addressing information necessary establish communication with the corresponding application entities. Access to this Information Base is provided by an Information Access Server and an corresponding Client. Collectively the Information Base, the Server and the Client provide an Information Access Service (LM-IAS). Both LM-IAS Client and Server entities are LM-MUX clients.

2.3.1 The Multiplexer

The LM-MUX provides a simple level of switching over the top of an IrLAP connection. It also hides the master/slave nature of IrLAP from the application and provides a symmetrical set of services to IrLMP clients.

The key goal here is to allow multiple independent sets of application entities to share access to the underlying IrLAP connection. This is increasingly important as the ability of portable platforms to multi-task improves. Also the mix of applications running on a portable platform tends to be chosen by the end-user from a potentially varied list of vendors. With this degree of 'end-user' integration it is simply not tenable to offer a solution that does not allow applications to share access to the IR media. For example: consider a file sharing application that allows portable device to access files on a desktop machine. This may result in a relatively long lived connection between the devices and the end user may not really be conscious of its existence. It would not be acceptable to have to shut down the file sharing software in order to then gain access to say an E-mail, printing or Fax services.

The functionality provided by LM-MUX has already been presented in Section 2.1 and Figure 2.1. LM-MUX also provides a device discovery operation that combines IrLAP device discovery and address conflict resolution into a single operation.

The introduction of a simple multiplexing function above the reliable device-to-device data transfer service of IrLAP does introduce one problem:

In general, multiplexing LM-MUX channels over a single IrLAP connection can lead either to data loss or deadlock. To illustrate consider a pair of peer application entities **A** and **B** connected by two LM-MUX connections. One LM-MUX connection is used to exchange data, while the other is used to send control. **A** sends on both connections and **B** receives on both connections.

Consider what happens if **A** sends a large amount of data to **B** while **B** attempts to read first from the control channel and then from the data channel. The following code fragments illustrate this behaviour[6]:

```
/* Behaviour of sender A */
for(...) {
    .
    .
  res = send(data_fd, data, sizeof(data));
    .
    .
}
res = send(control_fd, ctrl,sizeof(ctrl);
```

```
/* Behaviour of receiver B */
  recv(control_fd, ctrl,sizeof(ctrl));
  recv(data_fd, data, sizeof(data));
```

Data inbound for **B** is not being read. At some point buffer space holding inbound data for **B** becomes exhausted. We could allow the IrLAP flow-control mechanism to pause the sending of data by **A**. However, if we do that there is then no way that the control information from **A** to **B** can now be sent to allow **B** to progress to reading the data. The system is deadlocked. This is a general problem for any system that multiplexes channels over a reliable channel.

Alternatively, we avoid this deadlock situation by allowing IrLMP LM-MUX to discard inbound data that it cannot deliver. However this destroys the reliable delivery property that IrLAP gives us.

Allowing the deadlock possibility is by far the greater of these two evils, so the designers of IrLMP took the view that LM-MUX may discard inbound data that it is unable to deliver. The LM-MUX data transfer service is therefore best effort rather than reliable. The only possible cause of packet loss in these circumstances is inbound congestion within an LM-MUX channel. This inbound congestion may becompletely avoided by the inclusion of a flow-control mechanism within the channel between application entities.

The IrDA offers two suggestions for addressing this problem: a variant of the ISO 8073 Class 2 Transport Protocol [8] named IrTP [9]; and a bared credit based flow-control scheme dubbed Tiny TP [10].

As an alternative to flow control it should also be noted that designers of LM-MUX clients may choose instead to implement a retransmission scheme to recover from data lost due to inbound congestion; and in some circumstances the loss of data could simply be tolerated (e.g. playback of audio data).

Since these three alternatives exist, the IrDA has not mandated the use of any one particular.

There is one last facet of the multiplexer that is worthy of note. The negotiation present within IrLAP means that there is a deterministic upper bound to the time between the reversals of the underlying IrLAP connection. Some application designers may wish to exploit this deterministic behaviour however the presence of multiplexed streams provided by IrLMP obscures any guarantees provided by IrLAP. LM-MUX provides a mode of operation that grants exclusive access to the underlying IrLAP connection to just one LM-MUX connection.

2.3.2 Information Access Services

So far we have described a relatively straight forward communications mechanism that supports multiplexed communication channels between a pair of devices. A key goal for the IrDA has been ease of use. Previous IR solutions have had a tendency to disappoint users because the burden of ensuring that compatible peers application entities are active at each end of the link has fallen on the end-user. The LM-IAS within IrLMP provides the means for an application entity to identify and locate a compatible peer entity.

The LM-IAS Information Base contains a number of simple objects. Each object is an instance of a given class and contains a number of named attributes.

The class of an object implies the nature of the application entity that it represents, the data transfer method(s), the semantics of the information stream exchanged between peer entities etc. It also scopes the semantics of the attributes contained within instances of the given object class.

Both class names and attribute names my be up to 60 octets long. Since the meaning of an attribute is scoped by the class of the enclosing object there is no strict requirement to administer the attribute name space. Object class names do need to be administered, however it is intended that with such a large name

[6] Assuming blocking send and recv operations

space some sensible conventions will ensure that class name collisions do not occur. For example, Object Class definitions defined by the IrDA all start with the root "IrDA:".

Whilst in general attributes are scoped by the class of the enclosing object some attributes are of such general utility that they may be regarded as having global scope. In a formal sense this requires that they are identically defined in all object classes that adopt the use of such global attributes. Typically global attributes arise in order to express an address within the IrDA environment. For example, IrLMP defines the attribute "IrDA:IrLMP:LsapSel" to identify the LM-MUX service access point of a directly attached application entity. IrTP defines the attribute "IrDA:IrTP:TsapSuffix" to carry the portion of the Transport service access point address that extends the 32-bit device address. Likewise Tiny TP defines the attribute "IrDA:TinyTp:LsapSel". Application entities advertise their accessibility via these mechanisms by the inclusion of the corresponding attribute.

There are three attribute value types:

- Integer: A 32 bit integer
- User Strings: Intended for presentation via a User Interface. Up to 255 octets in length with multilingual support.
- Octet Sequence: An opaque sequence of up to 1024 octets of information. The attribute may impose further structure on the contents of the sequence. This is a good way to cluster a body of information under one attribute.

The IrDA requires that every IrDA compliant device provides a "Device" object that carries a long form of the device name and an indication of the version of IrLMP implemented on the device and the optional features that have been implemented. The long device name is useful as it allows names of up to 255 octets in length whereas the nickname exchanged during device discovery is restricted to 19 octets (<10 characters if Unicode is used to encode the nickname).

Access to a remote IAS Information Base is provided by a local IAS client entity that communicates with an IAS server entity on the remote device. The IAS server is statically bound to LSAP 0x00 on the multiplexer. This is the ONLY fixed address in the IrDA environment. All other application services are located by inspection of the Information Base. The IAS Client and IAS Server entities provide a number of querying operations on the Information Base. Get Value By Class is the only mandatory operation that both must support. This provides a 'shot-in-the-dark' mode of retrieving attributes from objects. The notion is that a client application entity knows what application service it seeks to make use of. For example a file sharing entity would be looking to make contact with a matching file sharing entity. It therefore knows the object class name of Information Base objects that represent such an entity and implicitly it knows the name and semantics of attributes attached to such objects. There is therefore little value in the application entity browsing the Information Base, it merely needs to attempt to retrieve known attributes from an instance of a known object class. This is precisely what Get Value By Class does.

The remaining optional IAS operations: Get Information Base Details; Get Objects; Get Value; Get Object Info and Get Attribute Names provide for richer interactions with the Information Base including the ability to browse the Information Base.

2.3.3 IrLMP Client Example

Consider a Fax Modem that offers independent data and control channels. The Fax modem advertises its data service by installing the following object in its local Information Base:

```
object 1 class FaxModemData {
  attribute IrDA:TinyTP:LsapSel =
        Integer(0x05);
}
```

The FaxModemData service makes use of Tiny TP and is accessible with an LM-MUX service access point selector of 5.

An client application entity that wishes to make use of the FaxModemData service performs the following operations:

```
IasValue *lsapSel;
DiscoverList *dl;
DeviceAddress *da;
FILE *fp;

/* Device Discovery */
dl = LM_DiscoverDevices(slots);

fp = NULL;

while (fp == NULL) {

/* Search Hints for Fax Device */
  while (dl != NULL) {
    if(dl->deviceInfo.hints & FAX_MASK){
      da = dl->deviceAddress;
      break;
     }
     dl = dl->next;
    }

/* Check for end of Discovery List */
  if(dl == NULL)
    break;

/* Read the LM SAP Selector */
  lsapSel = LM_GetValueByClass(
                da,"FaxModemData",
                "IrDA:TinyTP:LsapSel");

/* Connect if we got an LM-MUX SAP Sel */
  if(lsapSel != NULL) {
    fp = TinyTP_Connect(lsapSel,...)
  }
  dl = dl->next;
}
```

2.4 Upper Layers

We have already made mention of both IrTP and Tiny TP. Both of these may still be regarded as part of the plumbing as they form part of the conduit between peer application entities.

- IrTP [9] is based on the ISO 8073 Transport Protocol Class 2 [8]. The main task of the IrTP specification is to describe the interpretation of Transport Service Access Point Addresses in an IrDA context and to provide a mapping from the ISO Network Service primitives used by 8073 to IrLMP LM-MUX service primitives.

  IrTP provides:

  Per transport connection flow control

  Segmentation and reassembly of arbitrary sized PDUs

  Graceful disconnect

  More multiplexing.

- Tiny TP [10] defines a credit based flow-control scheme and relies on LM-MUX for multiplexing.

  Tiny TP provides:

  Per transport connection flow control

  Segmentation and reassembly of arbitrary sized PDUs

There is also scope for the development of transport protocols that exploit the deterministic behaviour provided by exclusive mode.

Members of the IrDA are currently working to define an mechanism for application level object exchange, OBEX [11]. Objects may be records from Personal Information Managers, diary entries, business cards etc.; Word processor, spreadsheet other traditional types of file; or other parcel of information e.g. an information hunting robot launched into a networkfrom say a PDA attached via IR to a payphone in an airport lounge.

2.5 Application Interfaces

Widespread implementation of the IrDA platform services described previously and the availability of consistent application programming interfaces (APIs) is the key to creating a market for IrDA aware applications.

The IrDA community is also currently working on two types Application Programming Interfaces (APIs) for IrDA. Legacy communications APIs and native IrDA APIs.

Legacy APIs

There is a general perception of Infrared as merely a cable replacement technology. From the preceding discussion it should be apparent that the inclusion of IrLMP, particularly LM-IAS, makes it much more. Nevertheless this perception persists and there is a desire to be able to run legacy serial and parallel port communications applications over Infrared in much the same way that terminal emulation applications were transitioned onto LANs from RS232 cables.

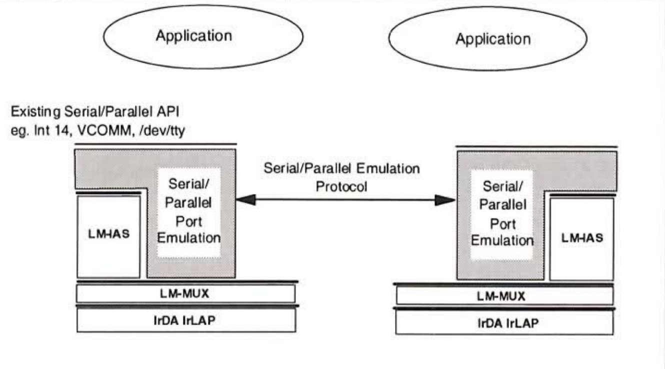

Application
Application
Existing Serial/Parallel API
eg. Int 14, VCOMM, /dev/tty
Serial/Parallel Emulation Protocol
Serial/ Parallel Port Emulation
LM-IAS
Serial/ Parallel Port Emulation
LM-IAS
LM-MUX
IrDA IrLAP
LM-MUX
IrDA IrLAP

Figure 2.5 Legacy Communications API Support

The IrDA has a Working Group known as IrCOM that is working on the emulation of the legacy communication interfaces typically provided by serial/parallel device drivers.

Native APIs

New applications will take full advantage of the potential that IrLMP offers to enable compatible application peers to identify and locate each other. This requires a native API for IrDA that exposes the full functionality of IrLMP, IrTP and Tiny TP to the application programmer.

IrDA members are interested defining a Winsock 2 Service Provider and API semantics [12][13]. Mapping the LM-IAS services into Winsock 2 is likely to prove a particular challenge!

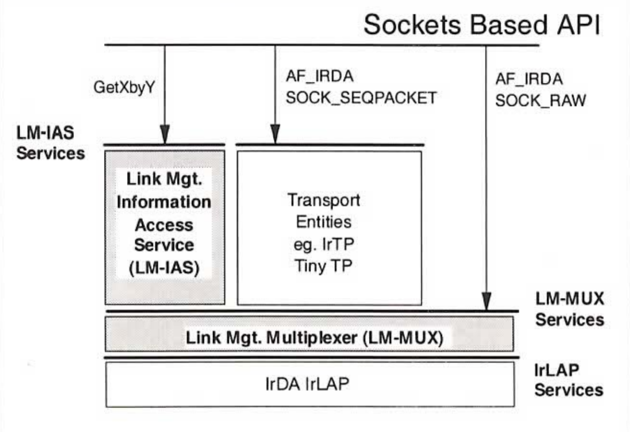


Figure 2.6 Sockets Based API

3. Application Services

With a stable platform on which to build application and application service designers can rapidly populate the space above the base platform. Some services will come about as the result of either open or closed collaborations, others will be the work of an individual or a single organisation. It is incumbent on application service designers to specify:

1. The IrLMP service hints that will be set if an instance of the service is being advertised.
2. An object class to carry the parameters essential to establish communication between peers.
3. Data transfer methods: i.e. raw, over IrTP, over Tiny TP or some other method specified in the service definition.
4. The application level protocol.

The degree to which the application service designer makes this information open is a matter of judgement for them. They may choose to seek endorsement from the IrDA; they may simply publish the specification of their service. Alternatively it may be regarded as proprietary and closed.

4. Summary

The future of short range directed infrared data communications looks bright!

With some 80 member companies the technology developed within the IrDA will soon be available on just about every significant mobile computing platform. End-users will be able to casually exchange, print and share information from whole documents to snippets from a diary or a business card. They will be able to do this without the hassle of needing to have the right cables and without having to do combat with pages of configuration and setup dialogues commonplace with serial and networked communication. Within the next year products with high speed IrDA interfaces will be readily available in the marketplace.

The ease of use that will be characteristic of IrDA applications is largely due to the device discovery and QoS negotiation facilities in IrLAP and the Information Access Services specified in IrLMP.

5. References

[1] P. D. Brown, L. S. Moore and D. C. York, "Low Power Optical Transceiver for Portable Computing Devices", U.S. Patent No. 5,075,792, Assignee: Hewlett-Packard Company, December 24, 1991.

[2] Infrared Data Association, "Serial Infrared (SIR) Physical Layer Link Specification", Version 1.0, April 27, 1994.

[3] International Organisation for Standardisation (ISO), "High Level Data Link Control (HDLC) Procedures - Elements of Procedures", ISO/IEC 4335, September 15, 1991.

[4] T. F. Williams, P. D. Hortensius and F. Novak, "Proposal for: Infrared Data Association Serial Infrared Link Protocol Specification", Version 1.0, IBM Corporation, August 27, 1993.

[5] Infrared Data Association, "Serial Infrared Link Access Protocol (IrLAP)", Version 1.0, June 23, 1994.

[6] Infrared Data Association, "Link Management Protocol", Version 1.0, August 11, 1994.

[7] HP/IBM/Sharp, "PROPOSAL: Fast Serial Infrared (FIR) Physical Layer Link Specification", Proposal to the Infrared Data Association, September 13, 1994.

[8] International Organisation for Standardisation (ISO), "Information technology - Telecommunications and information exchange between systems - Open Systems Interconnection - Protocol for providing the connection-mode transport service", ISO/IEC 8073, December 15, 1992.

[9] Infrared Data Association, "Use of ISO 8073 as an IrDA Transport Protocol", Version 1.0, August 12, 1994.

[10] Infrared Data Association, "TinyTP: A Flow-Control Mechanism for use with IrLMP", Version 0.1a, January 16, 1995.

[11] Infrared Data Association, "Object Exchange Protocol", Version 0.1a, January 4, 1995.

[12] Winsock 2, "Windows Sockets 2 Application Programming Interface: An Interface for Transparent Network Programming Under Microsoft Windows", Revision 2.0.6, February 1, 1995.

[13] Winsock 2, "Windows Sockets 2 Service Provider Interface: A Service Provider Interface for Transparent Network Programming Under Microsoft Windows", Revision 2.0.6, February 1, 1995.

6. IrDA Contact Information

Further information about the IrDA and copies of the IrDA Standards may be obtained from:

John Laroche
Executive Director
Infrared Data Association
P.O. Box 3883,
Walnut Creek
Califonia
USA 94598

Tel: (510) 943 6546
Fax: (510) 943 5600

E-mail: jlaroche@netcom.com

7. Author Information

The authors may be contacted at

Hewlett-Packard Laboratories, Bristol
Filton Road
Stoke Gifford
Bristol, BS12 6QZ
United Kingdom

Tel: +44 (117) 979 9910
Fax: +44 (177) 922 8920

E-mail: skw@hplb.hpl.hp.com
im@hplb.hpl.hp.com

Infrared Data Communications with IrDA

Charles D. Knutson, Ph.D.
Vice President, Research and Development
Chair, IrDA Test and Interoperability Committee

Glade Diviney
Manager, Test Tools and Services

Counterpoint Systems Foundry, Inc.
Corvallis, Oregon
knutson@countersys.com

Abstract

IrDA infrared communication is an inexpensive and widely adopted short range wireless technology that allows devices to "speak" easily to each other. Key protocol features make operation simple even for inexperienced users or devices with very little user interface. Digital cameras, phones, pagers, data collectors, set-top boxes, modems, kiosks, instruments, machinery, ID badges, watches, and computer peripherals are some of the natural users of this technology. This paper introduces IrDA infrared data communications and explores both mandatory and optional IrDA protocol layers and strategies.

1.0 Introduction

The Infrared Data Association was formed to enable universal point and shoot infrared connectivity between devices of all types. Today there are hundreds of devices that implement IrDA communication protocols and the dream of ubiquitous data transfer is becoming more of a reality. This paper briefly describes IrDA technology from low-level physical layers up to high-level optional protocols. It also describes IrDA Lite, a set of strategies for implementing minimal solutions for embedded systems.

2.0 The Infrared Data Association (IrDA)

The Infrared Data Association (IrDA) was formed in June, 1993. At IrDA's charter meeting, fifty companies came together to agree upon standard methods for communicating data via short range infrared transmission. Since that time, more than one hundred additional companies have joined IrDA, and hundreds of devices are currently available that implement IrDA communication protocols.

IrDA is administered by an Executive Director (John LaRoche) and an executive staff. The work of IrDA is conducted through three committees, whose chairs are elected each year. The Technical Committee is currently chaired by Dave Suvak of Counterpoint Systems Foundry, Inc. This committee refines and extends the hardware and software standards for IrDA. All new technical proposals come through this committee and its working groups. The Marketing Committee is currently chaired by Brian Ingham of IBM. This committee handles the marketing and promotional concerns of IrDA. The Test and Interop Committee is currently chaired by Charles Knutson of Counterpoint Systems Foundry, Inc. This committee deals with test specifications for hardware and software standards, as well as issues concerning the spread of interoperability and customer "out of the box" experience. The bulk of the work in IrDA is performed by special interest groups (SIGs) and working groups that carry specific charters.

3.0 The Promise of IrDA connectivity

One of the earliest motivations of the companies involved in IrDA was to eliminate wires and connectors with their accompanying limitations. Wires fray, wear, break, corrode, get tangled, and sometimes fail to reach far enough. The connectors on wires come lose, break, or otherwise become mangled and unusable. Wired connections clog desks with spaghetti-like entanglements, and are notoriously forgotten by the portable traveler who only later discovers that his mobile computer is useless without that one special wire that was left behind.

The core IrDA protocols were designed to replace wires with a "virtual wire" and some ability to accesses services over it. Originally, the focus of IrDA was to deliver this level of connectivity, and then let manufacturers worry about specific implementations above that. Since that time, it has become obvious that once the physical connector is gone (leaving in essence a "universal" connector), standardizing on higher level protocols can provide even greater levels of interoperability in the IrDA user space. Subsequently, a number of optional protocols have been approved, most of them derived from a specific vertical application model. Section 5.0 of this paper discusses the mandatory protocol layers of IrDA. Section 6.0 describes the optional higher-level protocols.

With these high-level protocols, a number of interesting and valuable use models are available to users. By standardizing on these protocols, application vendors can build systems that interoperate with systems of other vendors. This approach is quite similar to what we now see happening in the World Wide Web. The low-level TCP/IP protocol was not sufficient to provide interoperability. Without some higher-layer protocol, there were myriad ways of moving information, with a few garnering more common use than others (like FTP). However, when HTTP began to be used by web browsers with a standard file format (HTML), the elements were in place for a usage explosion. Suddenly anyone could build a web site, because they understood the file format. Anyone could access a web site using the common protocol. And anyone could build software to access anyone else's web site. When users had the "Ah ha!" experience of what the web could do, independent entrepreneurs did the rest and the web exploded, to the benefit of users. That is the power of universal data access. In the short range, walk up, point and shoot space, IrDA offers that same promise, which is becoming more and more a reality.

4.0 High-Level Overview

The IrDA protocols are organized in a traditional layered or stacked architecture. These individual layers are described in the next two sections. Some of these layers are required for a device to carry the IrDA logo. These are treated in Section 5. In addition, there are optional layers that apply to specific use models. These are discussed in Section 6.

The current protocols provide connectivity at distances up to one meter, and at speeds up to 4 Mbps. IrDA is interested in extending both of these limitations, and is currently working on extending specifications in both cases.

In a typical scenario, a user might have a PDA that has phone and address lists. The user might walk up to another PDA user and beam selected items using IrDA's IrOBEX protocol. Two Palm III users might discover that they each have games that the other does not have. These can also be beamed to the other device on the spot. A laptop user might want to print a document, but lacks a parallel cable. With IrDA capability on both the laptop and the printer, the laptop selects the appropriate LPT port (the one redirected to the IrDA port) and prints. These and other use models extend to devices of all types, including digital cameras, LAN access devices, pagers, cell phones, laptops, PDAs, printers, scanners, medical devices, etc. Some of these will be discussed in the individual discussions in Section 6 where the protocol relates to a specific use model.

In the basic IrDA use model, there are two devices. One is the primary and the other is the secondary. The primary device is responsible for selecting a device within its visual space, establishing a connection, and maintaining the virtual wire or link. The secondary responds when spoken to. At the beginning of a typical IrDA operation, the primary initiates a process known as "discovery", in which it explores its visible space for devices. From those devices that respond the primary selects a device and attempts to connect to it. During connection establishment, the two devices negotiate to understand each other's capabilities. In this way a connection can be optimized despite the unpredictable differences between two disparate devices. Once they have negotiated, they will jump to their highest common transmission speed, and attempt to communicate in ways that optimize the throughput and reliability of their connection.

Having established a connection, the devices may now search the services of the other devices. If the other device supports a desired service, a connection can be made to the service. At this point, applications on either side of the connection can transfer data. Obviously there are considerably more details than have been presented here, and the IrDA specifications are the definitive source for that information.

5.0 Required Layers

In this section, we explore the required IrDA layers, starting from the bottom and working our way up. Each of these layers is described in painstaking detail in corresponding IrDA specifications. The references for these documents are provided in Section 9.

Figure 1 shows the basic organization of the IrDA stack.

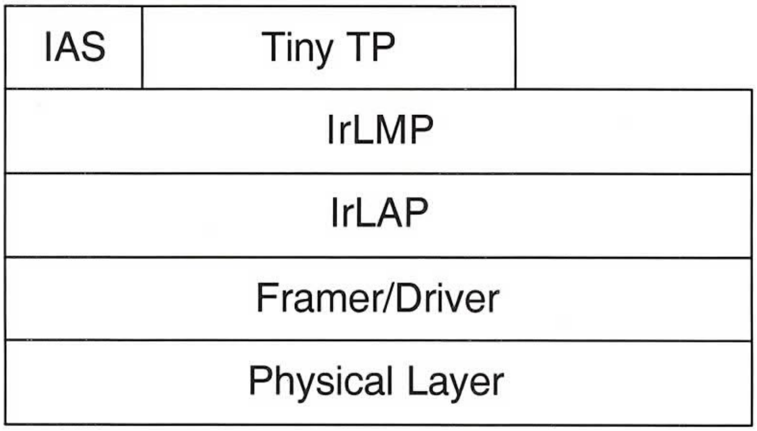


Figure 1. IrDA protocol stack layers.

5.1 Physical Layer

IrDA transceivers broadcast infrared pulses in a cone that extends from 15 degrees half angle to 30 degrees half angle off center. The IrDA physical specifications require that a minimum irradiance be maintained so that a signal is visible up to a meter away. Similarly, the specifications require that a maximum irradiance not be exceeded so that a receiver is not overwhelmed with brightness when a device comes in close. In practice, there are some devices on the market that do not reach one meter, while other devices may reach up to several meters. There are also devices that do not tolerate extreme closeness. The typical sweet spot for IrDA communications is from 5 cm to 60 cm away from a transceiver, in the center of the cone.

IrDA data communications operate in half-duplex mode. The reason is quite simple. While transmitting, a device's receiver is blinded by the light of its own transmitter. Because of this, full duplex communication is not feasible. The two devices that communicate simulate full duplex communication by quickly turning the link around. The primary device controls the timing of the link, but both sides are bound to certain hard constraints and are encouraged to turn the link around as fast as possible.

Transmission rates fall into three broad categories: SIR, MIR, and FIR. Serial Infrared (SIR) speeds cover those transmission speeds normally supported by an RS-232 port (9600 bps, 19.2 Kbps, 38.4 Kbps, 57.6 Kbps, 115.2 Kbps). Since the lowest common denominator for all devices is 9600 bps, all discovery and negotiation is performed at this baud rate. MIR (Medium Infrared) is not an official term, but is sometimes used to refer to speeds of .576 Mbps and 1.152 Mbps. Fast Infrared (FIR) is deemed an obsolete term by the IrDA physical specification, but is nonetheless in common usage to denote transmission at 4 Mbps. "FIR" is sometimes used to refer to all speeds above SIR. However, different encoding approaches are used by MIR and FIR, and different approaches are used to frame MIR and FIR packets. For that reason, these unofficial terms have sprung up to differentiate these two approaches. The future holds faster transmission speeds (currently referred to as Very Fast Infrared, or VFIR) which will support speeds up to 16 Mbps.

5.2 Framer/Driver

The framer and the driver are actually two separate functions, but have enough in common that they are typically grouped together (and commonly referred to simply as "framer"). The driver portion refers to the software that acts as a device driver for the system's transceiver controller. This driver initializes the infrared hardware, changes transmission speeds, delivers data to the transceiver, and receives data from the transceiver.

The framer portion refers to the bundling of a data packet into a form that can be given to the hardware. This may include the calculation of cyclic redundancy check value, the addition of start and stop bits, and transparency for reserved bytes. Because the framing approach varies with the transmission speeds, it is most common for the framer and driver functions to be combined. In this way, all hardware dependencies in a system can be localized to one section of the IrDA stack.

5.3 IrLAP: Link Access Protocol

The IrDA Link Access Protocol (IrLAP) is responsible for performing device discovery and negotiation, and for preserving the physical connection, or "virtual wire". It is at this level that the concept of primary and secondary devices is relevant. IrLAP is based on HDLC, adding features to facilitate the walk up, nature of IrDA connections.

IrLAP provides a reliable transmission medium on which to build additional communications. It facilitates error detection, retransmission of lost or damaged packets, and rudimentary flow control.

5.4 IrLMP: Link Management Protocol

The IrDA Link Management Protocol (IrLMP) allows one or more IrDA services to run over a single IrLAP connection. Applications using an IrDA stack can read/write directly to IrLMP, or can use other higher level protocols that, in turn, read/write to IrLMP. A typical service running on IrLMP might include the printing application on an IrDA-enabled laser printer. This application would register its service with IAS (described below), and then be able to service a print job via the IrLMP connection, should a user walk up and begin accessing the printer via the IrDA port.

5.5 IAS: Information Access Service

The Information Access Service (IAS) is the only required service available through IrLMP. It is the mechanism by which applications advertise and access services. Applications register their services when they load, and are given a specific selector, called an LSAP (Link Service Access Point) Selector, by which the service can be accessed by other devices. There are no pre-defined LSAP Selectors besides the IAS itself at selector 0. When a device connects to another, an application can query the IAS of the other device to determine what services it might have, and on which LSAP Selectors. Once the LSAP Selector is known, the application can connect and begin data transfer.

5.6 Tiny TP: Tiny Transport Protocol

Tiny TP is a transport protocol that provides two basic services: flow control and segmentation and reassambly (SAR). Tiny TP allows flow control per service channel, where the more rudimentary flow control provided by IrLAP controls the entire physical link. Flow control in Tiny TP is credit based, permitting an application to extend enough credit to the other side so that it won't be overwhelmed. The remote device uses these credits as packets are delivered. More credits are extended as space becomes available to receive. Segmentation and reassembly provides a mechanism for delivering large packets to the IrDA stack, allowing Tiny TP to break up packets (segmentation) on one side and, on the other side, put them back together (reassembly). This approach takes the burden off applications to be concerned with IrDA packet sizes. This feature is particularly relevant for IrLAN.

6.0 Optional Layers

The following layers are all high-level protocols, and are not strictly required by IrDA. However, some of them are essential within the context of certain use models, in the same way that HTTP is essential for web access, even if it's not part of a basic TCP/IP protocol stack.

6.1 IrOBEX: IrDA Object Exchange

IrDA Object Exchange (IrOBEX) can be viewed as essentially "HTTP for IrDA". IrOBEX was designed to resemble HTTP, and it leverages what it can from this internet protocol, adding capabilities that relate to the unique environment of IrDA. IrOBEX is best used in situations where objects of some kind need to be moved from one device to another. For example, two devices may exchange phone and address information, or calendar information in vCard and vCal formats. Or, a handheld scanner may capture a graphics image and beam it to a laptop to manipulate. Both of these are classic uses for IrOBEX. Because of its universal applicability for object movement, where applicable, IrOBEX is a required protocol for devices seeking interoperability certification.

6.2 IrCOMM

IrCOMM is designed to provide legacy support for applications that already run over COM ports. For example, assume we have a PDA with a cradle that plugs into the serial port of a computer. The desktop software for this PDA is designed to communicate using a serial cable connected to the PDA cradle. To allow synchronization directly between the computer and the PDA, the PDA could be enabled with IrCOMM. Then, by selecting a virtual COM port, the synchronization could take place over infrared without introducing any changes to the computer's software. This is an example of a legacy application for IrCOMM. While this works in legacy situations, IrCOMM is strongly discouraged as a platform for developing new use models, since it reduces IrDA's rich feature set to a virtual nine wires, requiring sophisticated applications to recreate many of the capabilities that are already present in the IrDA stack.

6.3 IrLPT

IrLPT is part of the IrCOMM specification, and is also referred to as IrCOMM 3-Wire Raw. It deserves special mention because it is the mechanism by which legacy printing is achieved between devices and IrDA-enabled printers. Support on desktops is achieved through a virtual LPT port that maps to the IrDA port. When an application or printer driver is configured with the virtual LPT port, infrared printing is enabled without changes to the printer driver or application. As with IrCOMM, IrLPT is intended for legacy support of existing applications. Because of its importance for legacy printing, where applicable, IrLPT is required for devices seeking IrReady interoperability certification.

6.4 IrTran-P

IrTran-P (IrDA Transfer Picture) represents a specific mechanism used by some manufacturers to transfer digital images between devices. IrTran-P is an IrDA application note, meaning that it represents a particular way of solving this problem, without carrying the mandate of IrDA as the only appropriate way to do it. IrTran-P is built on IrCOMM, and therefore requires the reconstruction of several key components to manage services and object exchange. Specifically, IrTran-P adds SCEP (Simple Command Execute Protocol) for service access and link management, and bFTP (Binary File Transfer Protocol) for digital image object exchange. In addition, IrTran-P defines its own digital image file format, UPF (Uni-Picture Format) so that IrTran-P devices can communicate effectively.

6.5 IrMC: IrDA Mobile Communications

IrMC (IrDA Mobile Communications) is a set of four protocols proposed by the IrDA's Mobile Communications Working Group. This group is fundamentally concerned with IrDA communication between telecom devices, such as pagers and cell phones. However, many of the features of IrMC are applicable to other devices, such as PDAs. Because of that, the scope of IrMC has expanded to include devices of all types. IrMC incorporates the following protocols: IrOBEX, IrCOMM, RTCON, and Ultra. IrOBEX, described in Section 6.1, is used in IrMC to exchange vCards, vCalendars, and similar objects. IrCOMM, described in Section 6.2, is used in IrMC to permit cellular phones to be used as external modems, via a virtual COM port connection between a laptop (or other device) and a cell phone. RTCON is described in Section 6.6. Ultra is a very small, connectionless communication mechanism that severely constrained devices can use for device programming and small object exchange using a connectionless version of OBEX.

6.6 RTCON

Real Time Transfer Control Protocol (RTCON) is used to transmit real-time voice and control data over an infrared link. In a typical use model, a cell phone can be placed in a cradle in a car, and an infrared link can be established between the speaker phone mechanism in the car and the cell phone. This permits dialing and talking without holding a cell phone while driving (which is illegal in some parts of the world—most notably Europe and Japan). The call is made with the cell phone, but the voice data in the phone is transferred via infrared to the in-dash speaker phone. This permits users to use their normal cell phones without having to have a special cell number specifically for their car (with accompanying duplicated fees).

6.7 JetSend

JetSend is a technology created and licensed by Hewlett-Packard for delivery of digital imaging information via a variety of transport mechanisms. The first two transports on which JetSend was implemented were TCP/IP and IrDA. JetSend permits devices to negotiate to their greatest common image handling capacity, eliminating the need for hardware-specific printer drivers. In a static, wired office environment, the accessible printers don't change that often. But for mobile devices such as laptops and PDA's, finding and installing the appropriate printer driver can be a significant problem. With JetSend, any IrDA enabled device can approach an IrDA-enabled JetSend printer, and render the best possible image.

7.0 IrDA Lite

Strictly speaking, IrDA Lite is not a protocol, but it is significant enough in the world of IrDA to deserve brief mention. The majority of devices incorporating IrDA are embedded devices. Most of these devices provide dramatically less memory than laptops or desktops. IrDA Lite renders the minimal implementation of IrDA that still interoperates with "full-featured" IrDA stacks. It does so by sacrificing speed, throughput, and non-essential features.

The effort would be similar to removing parts from a car, but requiring that it still seat one passenger and be sufficiently capable of traveling some distance under certain conditions. For the use models where memory savings are worth the loss in throughput, the fit is great.

Typical IrDA Lite strategies include limiting the packet size to 64 bytes, limiting window size to one, limiting transmission speed to 9600, and using a simplified state chart. By employing these and other IrDA Lite strategies, it is possible to achieve a two to five fold reduction in RAM and ROM requirements. For some small devices, throughput and features are not as important as memory, and the decision to use IrDA Lite is easy. For other devices, the tradeoffs are not so straightforward. In these cases, one need not employ all IrDA Lite strategies. A designer can employ those strategies that make the most sense, garnering the memory savings desired, without completely sacrificing throughput or feature set.

8.0 Summary

The number of IrDA devices available has been growing rapidly for the last five years and its growth continues to accelerate in the portable device market. The vision of ubiquitous point and shoot connectivity is becoming more commonplace. The continued spread of IrDA technology will be a key enabling factor in broadening acceptance of portable devices of all types and sizes throughout the world.

9.0 List of IrDA Specifications

IrDA Serial Infrared Physical Layer Link Specification, v1.2, November 10, 1997

IrDA Serial Infrared Link Access Protocol (IrLAP), v1.1, Jun 16, 1996

IrDA Link Management Protocol, v1.1, January 23, 1996

IrDA Tiny TP: A Flow Control Mechanism for Use with IrLMP, v1.1, October 20, 1996

IrDA IrCOMM: Serial and Parallel Port Emulation over IR, v1.0, November 7, 1995

IrDA Object Exchange Protocol (IrOBEX), January 22, 1997

Minimal IrDA Protocol Implementation (IrDA Lite), v1.0, November 7, 1996

IrDA Serial Infrared Physical Layer Measurement Guidelines, v1.0, January 16, 1998

IrDA Interoperability Test Plan and Process, v1.2, September 28, 1998

The IrDA Standards for High-Speed Infrared Communications

Iain Millar

Martin Beale

Bryan J. Donoghue

Kirk W. Lindstrom

Stuart Williams

As more data communications products, such as printers and laptop PCs, are released with infrared capability, support for a core set of IrDA standards has strong support from many manufacturers because, among other things, they want to ensure that their products will interoperate in a transparent and user-friendly manner.

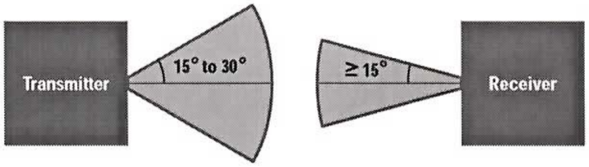
The use of infrared techniques for data communications has been around for several years, and by 1993 several commercial products were available with this capability. However, each company has tended to have its own infrared standard, and although devices from the same manufacturer could communicate with each other, competing systems tended not to be interoperable. Examples of such proprietary infrared systems include Hewlett-Packard's HP SIR (serial infrared), Sharp's ASK systems, and General Magic's MagicBeam. The resulting confusion in the marketplace meant that users viewed infrared as having only limited utility.

On June 28, 1993, the Infrared Data Association (IrDA) had its first meeting with the purpose of establishing a ubiquitous, low-cost, point-to-point serial infrared standard. Some 50 representatives from 20 interested companies were expected, but over 120 people representing more than 50 companies actually attended. It was clear that the industry was interested in developing a standard that would allow the true value and utility of infrared to be realized. At the culmination of this process—and due in no small part to the enthusiasm and spirit of cooperation of the participating companies—the first IrDA standards were published, just one year and two days after the initial meeting.

To date, IrDA has specified the physical and protocol layers necessary for any two devices that conform to the IrDA standards to detect each other and exchange data. The initial IrDA 1.0 specification detailed a serial, half-duplex, asynchronous system with transfer rates of 2400 bits/s to 115,200 bits/s at a range of up to one meter with a viewing half-angle of between 15 and 30 degrees (see **Figure 1**). More recently, IrDA has extended the physical layer specification to allow data communications at transfer rates up to 4 Mbits/s.

Figure 1

Viewing angle specified in IrDA specification 1.0.



This paper presents details of the individual IrDA specifications, focusing specifically on the high-speed extensions that allow data communications at up to 4 Mbits/s. The first section gives details of the objectives that resulted in the series of IrDA specifications. The specifics of the user model and the technical requirements of the specification are also presented. Next the IrDA architecture is introduced, highlighting how the IrDA specifications together provide overall functionality. The infrared physical-layer specification with particular emphasis on modulation format, packet framing, transceiver design, and clock recovery is discussed in the next section. The transceiver design for the HP HDSL-1100 IrDA transceiver is also described in this section. The last section covers the protocol layers of the IrDA specifications. Finally, IrDA's current status is summarized.

IrDA Objectives

When IrDA was established, it set for itself the following objective:

"To create an interoperable, low-cost infrared data interconnection standard that supports a walk-up, point-to-point user model* that is adaptable to a broad range of mobile appliances that need to connect to peripheral devices and hosts."[1]

IrDA chose the short-range, walk-up, point-and-shoot directed infrared communications model for two main reasons. First, it was perceived that the initial target market for IrDA-enabled devices would be the mobile professional who is also a computer user. The environment for the use of such devices would be in a typical working environment in which the majority of stationary devices, such as printers or computers, would be located within the user's own reach space, on the desktop or in the immediate vicinity. Typical use of such devices would consist of short, conscious interactions such as file transfer or printing. Such use scenarios do not require the devices to be continually connected to each other, and a directed model of communications was adopted in which the user consciously points the infrared device at the target.

Previously, mobile professionals might connect their laptops to various peripherals using parallel or serial cables. Connecting such devices using LAN connections might also be possible if cost were not an issue. However, a problem arises when the user becomes mobile—for example, when visiting customers in their office. Setting up a laptop at the customer office to achieve even simple tasks, such as printing or file transfer, would more than likely require significant reconfiguration. IrDA aimed to change this by providing standards for ubiquitous access to such devices.

Second, IrDA chose this communications model to minimize cost. The use of a single LED and photodiode in the transceiver enables an extremely low-cost implementation. The model simplifies the protocol software by restricting the number of visible devices, hence limiting the contention and interference between IrDA devices. The limited range also allows reuse of the infrared medium, allowing multiple pairs of devices to communicate at the same time.

* The phrase "walk-up, point-to-point user model" refers to the fact that to ensure data transfer between devices with infrared capabilities, they must be placed close together (< 2 m) with their infrared transceivers pointed at one another.

Glossary

Cell. A symbol in PPM.

Chip. A pulse within a symbol (cell) in PPM.

ENDEC. The encoder-decoder used in the IrDA physical layer.

HTTP Hypertext Transfer Protocol.

HDLC. A bit-oriented, synchronous High-level Data Link Control protocol that applies to the message-passing (data link) layer of the Open Systems Interconnect (OSI) model for computer-to-computer communications.

IAS. The information access service maintains information about the services available on the host device and provides services that allow access to information on remote devices.

IrCOMM. IrDA specification for the emulation of serial and parallel port communications.

IrLAN. IrDA specification for accessing a LAN over an infrared medium.

IrLAP. IrDA specification for Link Access Protocol. This document specifies an HDLC-based protocol for controlling access to the infrared medium.

IrLMP. IrDA specification for Link Management Protocol. This protocol provides the LM-MUX and LM-IAS services.

IrOBEX. IrDA specification that defines the protocol for generic object exchange in an IrDA-enabled device.

IrPHY. The specification that describes the physical layer properties of the IrDA standard.

LM-IAS. The Link Management Information Access Service allows a pair of IrDA devices to interrogate each other to determine the services available on each device.

LM-MUX. The Link Management Multiplexer allows any pair of IrDA devices to simultaneously and independently use a single IrDA connection between themselves.

LSAP. Link Service Access Ports are address fields that uniquely identify applications on the source and destination devices.

LSAP-SEL. Link Service Access Port Selector.

PPM. Pulse position modulation.

SIR. Serial infrared.

Tiny TP. Lightweight transport protocol specification.

IrDA aimed to allow its standards to support a wide class of computing devices and peripherals that might be used by mobile professionals. These devices would range from very sophisticated, high-power notebook or laptop personal computers, through palmtop computers and personal digital assistants, to simple single-function devices like electronic business cards or phone dialers. Target peripheral devices would include conventional computer-oriented devices like printers and modems, as well as automatic teller machines and public and mobile telephones. It was also envisaged that IrDA would enable new classes of devices such as information access points.

To target such a broad range of devices, a set of general requirements was placed on any prospective standard. These requirements included:

- Low cost
- Industry standard
- Compact, lightweight, low-power

- Intuitive and easy to use
- Noninterfering.

Using these requirements, the IrDA committee developed a series of standards aimed at providing ubiquitous, low-cost, directed infrared communications for all classes of mobile computing devices. In IrDA's vision of the world, the user of such devices would be able to roam across international boundaries using IrDA communications to access information, computing, and communications services in a uniform and transparent manner. The days of the mobile computer user travelling the globe with a multitude of modem, serial, and parallel cables, including adapters, will be gone.

The remainder of this paper presents details of the standard IrDA has put in place to achieve this vision.

The IrDA Architecture

After the initial marketing requirements had been specified, the technical committee within IrDA moved quickly towards the development of the initial standards. In April 1994, the first IrDA standard was published covering the physical layer properties. This document, the Infrared Physical Layer (IrPHY) specification,[2] describes an infrared transmission system based on a UART modulation strategy. The document specifies the necessary parameters to provide an asynchronous half-duplex serial communications link over distances of at least one meter at data rates between 2400 bits/s and 115.2 kbits/s. The cone half-angle of the infrared transmission is specified as being at least 15 degrees, but no more than 30 degrees. The IrPHY specification was quickly followed with the publication of the Infrared Link Access Protocol (IrLAP) in June 1994.[3] IrLAP specifies an HDLC-based protocol for controlling access to the infrared medium and providing the basic link-level connection between a pair of devices.

During the development of IrPHY and IrLAP, it was realized that some additional functionality was required in addition to the ability to provide a single connection between a pair of devices. The Infrared Link Management (IrLMP) layer was conceived.[4] This layer has two primary functions.

First, it provides the mechanism by which multiple entities within any pair of IrDA devices can simultaneously and independently use the single IrLAP connection between those devices. This function is called the link management multiplexer (LM-MUX).

Second, it provides a way for entities using the IrDA services to discover what services are offered by a peer device and to register available services within the local device. This link management information access service (LM-IAS) considerably benefits the ease of use of portable devices, allowing pairs of devices to interrogate each other to discover information about the applications within each device.

These three standards—IrPHY, IrLAP, and IrLMP—form the core of the IrDA architecture, and all are required for a device to be IrDA-compliant. Since the core documents were published, several extensions have been added. The current complete IrDA architecture is shown in **Figure 2**.

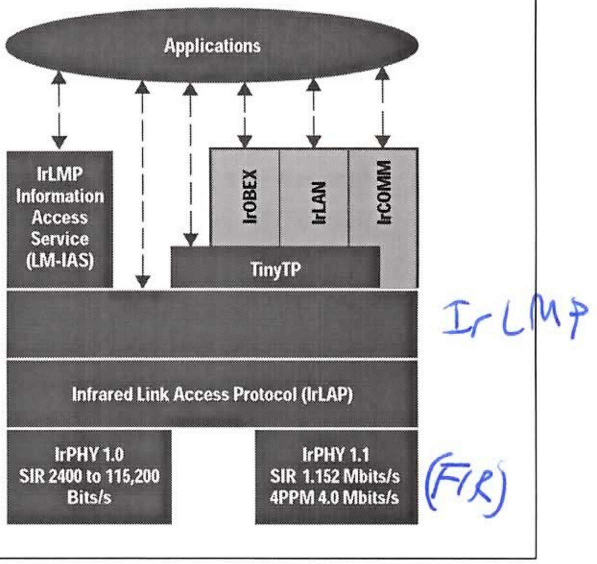
In October 1995, optional extensions to the physical layer, adding data transmission speeds of up to 4 Mbits/s, were accepted by the IrDA committee. These changes resulted in the IrDA IrPHY 1.1 specification.[5] The IrLAP and IrLMP documents have also recently been updated to version 1.1 to incorporate various improvements that resulted from practical experience in implementing and using the IrDA protocols.[6,7]

In addition to the base standards, IrDA has specified a protocol called Tiny TP.[8] This protocol is an extremely lightweight transport protocol designed to provide application-level flow control as well as segmentation and reassembly of application data units. This protocol has proved to be useful and is now implemented by most applications that support the IrDA architecture.

To complement the functionality of the main components of the IrDA architecture, several application-level protocols have been and are in the process of being developed. These protocols are aimed at providing convenient and uniform interfaces to the functionality of the IrDA protocols for both old and new applications.

Figure 2

The IrDA architecture.


Applications
IrLMP Information Access Service (LM-IAS)
IrOBEX
IrLAN
IrCOMM
TinyTP
IrLMP
Infrared Link Access Protocol (IrLAP)
IrPHY 1.0 SIR 2400 to 115,200 Bits/s
IrPHY 1.1 SIR 1.152 Mbits/s 4PPM 4.0 Mbits/s
(FIR)

The original target for IrDA was cable replacement. The need for a protocol to support the redirection of serial and parallel cable traffic resulted in the IrCOMM serial and parallel port emulation protocol specification.[9] This protocol enabled the redirection of conventional serial and parallel ports over the infrared medium, allowing many existing applications to operate unchanged over an IrDA link. Another area seen as a suitable application of IrDA, particularly as a result of the high-speed extensions, is wireless access to local area networks. The protocol IrLAN was developed to allow an IrDA-enabled device to access a LAN over the infrared medium.[10] The protocol, in combination with an IrLAN-compatible LAN access device, provides the IrDA device with the equivalent functionality of a LAN card and the advantages of wireless connectivity.

Both IrCOMM and IrLAN address legacy-style applications. However, it is envisioned that many new applications will be enabled by the IrDA standards. Using IrDA on low-end devices gives rise to the need for a flexible, lightweight information exchange protocol suitable for devices with varying resource capabilities. A protocol for generic object exchange, IrOBEX, is currently under development within IrDA.[11] This protocol is based on HTTP (Hypertext Transfer Protocol) but is more compact. When completed, IrOBEX will provide a device independent method for exchanging arbitrary units of data between IrDA-enabled devices.

The IrDA Physical Layer

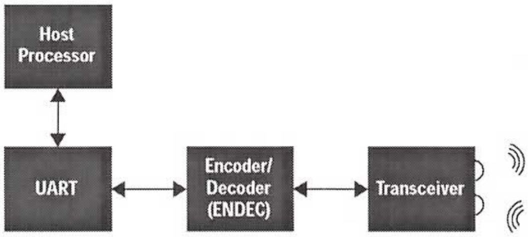
The IrDA physical layer is split into three distinct data rate ranges: 2400 to 115,200 bits/s, 1.152 Mbits/s, and 4 Mbits/s. Initial protocol negotiation takes place at 9600 bits/s, making this data rate compulsory. All other rates are optional and can be added if a device requires a higher data rate. The links are designed to be used in a line-of-sight, point-and-shoot manner and hence have a modest minimum coverage of one meter, with a ±15° viewing angle. This modest coverage is advantageous, since it allows a low-cost, high-data-rate link to be produced in a small package.

2400-to-115,200-bit/s Link

This is based on the HP-SIR link developed for HP calculators.[12] All IrDA-compliant devices implement this type of link since initial protocol negotiation takes place at 9600 bits/s. The architecture of the link (**Figure 3**) is designed for easy implementation and low cost. Hardware costs can be kept to a minimum by implementing the protocol, packet framing, and CRC calculation in software on the host processor. Bytes of data from the processor are converted to a serial data stream by a UART (universal asynchronous receiver-transmitter). Since many systems already include a UART for RS-232 communications, this places no extra cost burden on the system. Only the ENDEC (encoder-decoder) and transceiver represent an additional hardware cost for the system.

Figure 3

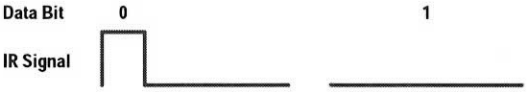
The 2400-to-115,200-bit/s link architecture.



Infrared receivers contain a high-pass filter to remove background daylight. This high-pass filter forces the use of encoding on the link to ensure that long strings of zeros or ones are not lost in transmission. The encoding used on this link is return-to-zero (RZ). Zeros are represented by a pulse of 3/16-bit duration, and ones by the absence of a pulse (**Figure 4**). For example, 3/16 of a pulse width at 115,200 bits/s is 1.6 μs. The code is power-efficient since infrared light is only transmitted for zeros. The tall narrow pulse has better signal-to-noise ratio performance than a short wide pulse of the same energy.

Figure 4

The coding on a 2400-to-115,200-bit/s link.



The 1.152-Mbit/s Link

At speeds above 115,200 bits/s, packet framing and CRC generation and checking become a significant burden to the host processor. At 1.152 Mbits/s, these tasks are performed in hardware by a packet framer (see **Figure 5**). The packet format is slightly different from that used in the 2,400-to-115,200-bit/s link, but the line code remains similar.[5] Higher-level protocols are less processor intensive than packet framing or CRC generation and are still implemented in software on the host processor.

The 4-Mbit/s Link

The 4-Mbit/s link architecture is shown in **Figure 6**. As in the 1.152-Mbit/s link, packet framing and CRC generation and checking are performed in hardware to relieve the burden on the host processor, while higher-level protocols are implemented in software on the host processor. The link uses a new encoding scheme (described below) and a new, more robust packet structure. A phase-locked loop replaces edge detection as the means of recovering the sampling clock from the received signal. The packet framer, ENDEC, and phase-locked loop are more complex than the UART and ENDEC in the 2400-to-115,200 bit/s link. However, this added complexity need not be expensive. The components are specified in a

Figure 5

The 1.152-Mbit/s link architecture.

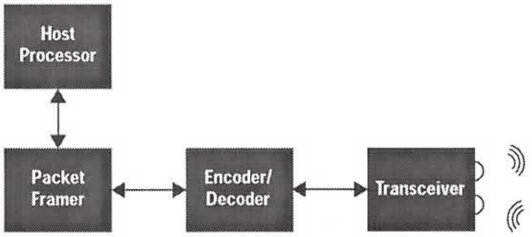
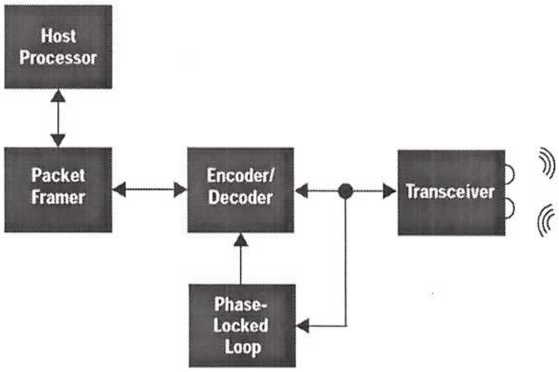


Figure 6

4-Mbit/s link architecture.

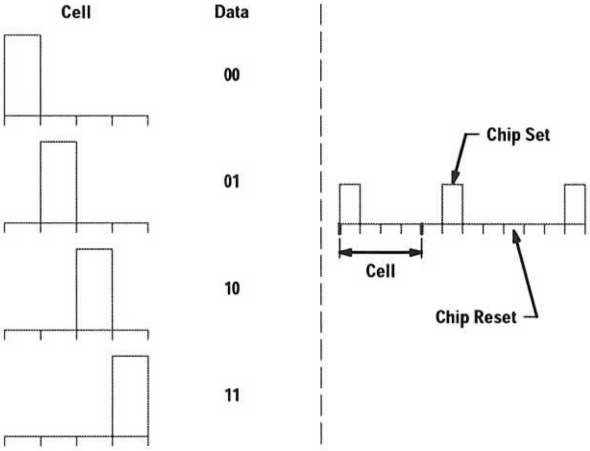


hardware description language and can be added quickly and inexpensively to one of the host system's ASICs. PC chipsets including the 4-Mbit/s hardware are already available from leading semiconductor manufacturers.

Coding and Packet Format. Pulse position modulation (PPM) was chosen as the line code for the 4-Mbit/s link. Data is transmitted within a PPM signal by varying the position of a pulse (referred to here as a chip) within a symbol (referred to here as a cell). The PPM modulation for the 4-Mbit/s link allows one chip to be set in one of four possible positions; thus it is known as 4PPM. Since a chip can be set in one of four possible positions, four different messages can be sent within one cell, allowing two bits of data to be encoded per cell. **Figure 7** shows the four possible messages that can be transmitted by 4PPM.

Figure 7

4PPM message encoding.



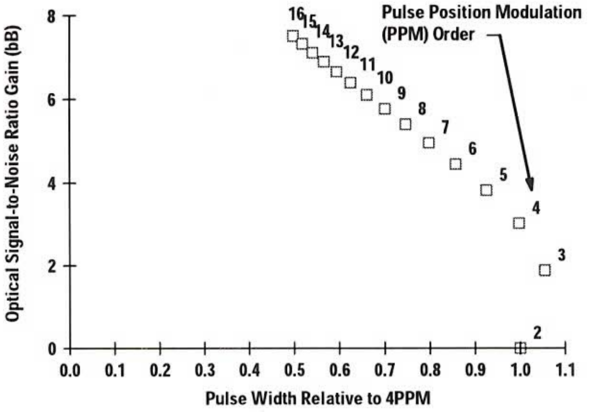
Pulse position modulation has many properties that make it attractive for use on the free-space optical channel. One of the main properties is the sparseness of the code. Sparse code allows high peak powers to be employed for set chips while maintaining a reasonable average power. The eye-safety rules stipulate a maximum average optical power, and LEDs tend to be average-power-limited at moderate duty cycles.

Pulse position modulation also contains significant and regular timing content, which facilitates synchronous clock recovery using a phase-locked loop. It is a modulation format that has very little dc content and can be high-pass filtered at 100 kHz, avoiding interference generated by fluorescent lighting without adversely affecting the receiver's eye diagram. A particularly interesting feature of PPM—one that had important ramifications in the choice of end delimiters—is its ability to detect line code errors.

Higher orders of PPM give lower duty cycles and theoretically greater signal-to-noise ratio gains on the infrared medium. **Figure 8** illustrates the interesting relationship between signal-to-noise ratio gain achievable with various orders of PPM and the required pulse width. It is interesting to note that the optimum order of PPM from a bandwidth efficiency perspective would be 3PPM. This result might be of theoretical interest, but is fairly useless in a practical system. Since the fastest bright LEDs have a rise time of around 40 ns, and the rise time of an LED is proportional to the pulse width, the use of high-order PPMs at 4 Mbits/s becomes impractical. The decision to adopt the order four for the PPM was motivated by knowledge of the range of duty cycles over which LEDs are peak-power-limited, the rise and fall time of available LEDs, and the frequent timing content provided at order four.

Figure 8

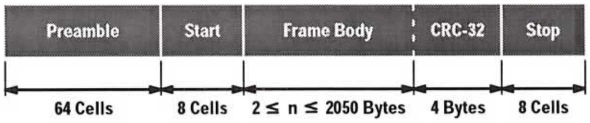
The signal-to-noise ratio gain and pulse width trade-off.



Packet Format. The 4-Mbit/s physical layer packet has distinct features that perform a useful and well-defined role (see **Figure 9**). A preamble allows dc balance to be attained, and more important, permits the phase-locked loop to achieve chip-level synchronization. The length of the preamble was considered carefully such that the preceding two goals could be achieved without a significant impact on efficiency. The start and stop delimiters provide cell and frame synchronization and were chosen so as not to compromise overall packet robustness or adversely affect the receiver eye diagram. To distinguish the preamble and the end delimiters from the frame body, these fields contain code violations. The body of

Figure 9

The 4-Mbit/s packet format.

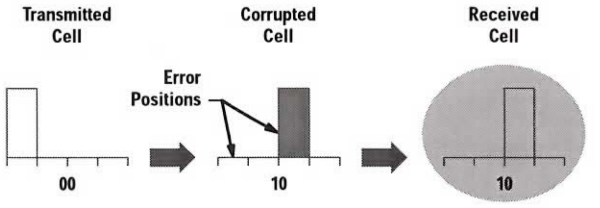


the packet is 4PPM-coded and has a 32-bit cyclic redundancy check (CRC) field appended to it. The choice of a 32-bit CRC provides a guaranteed level of robustness to undetected data errors over the range of error rates expected on a free-space infrared channel. The CRC is performed on the data bits rather than on the PPM-encoded chips.

Error Detection and Delimiters. A decoder may choose to exploit the error detection capabilities of 4PPM. The only portions of the packet allowed to contain violations are the preamble and the frame delimiters. If a decoder finds code violations within the frame body or CRC portion of the packet, it can flag that packet as being corrupted. In the same way that a sufficient number of carefully positioned errors can produce a correct-looking CRC for a corrupted packet, there are some error patterns that a 4PPM decoder cannot detect. An example is shown in **Figure 10**.

Figure 10

An undetectable 4PPM error.



Because of some spurious interference, cell data is erroneously changed from 00 to 10.

The corrupted cell obeys PPM rules, but the error is undetectable by the 4PPM cell decoder.

The role of the CRC is to detect those error patterns that the PPM cell decoder cannot detect. Owing to the combined distance structure of the CRC and the pulse position modulation, the packet can be made very robust to withstand either random or burst errors at any signal-to-noise ratio.

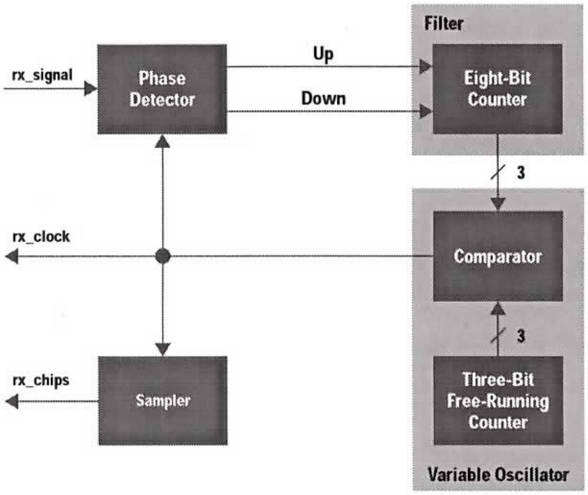
A more worrisome error mechanism that had to be considered was the possibility of the corruption of the frame delimiters. The frame delimiters are not in themselves protected by the CRC. If the situation arose whereby a false Stop delimiter appeared in a valid position within the data and CRC portion of the packet, the packet would be protected solely by the scrambling effect of the CRC. In this case, a corrupted packet would be flagged as correct with a probability of $(0.5)^{32}$. Thus, it is important to ensure the unlikelihood of either random or burst errors causing a false delimiter to appear within the data portion of the packet. This is achieved by choosing delimiters with a large Hamming distance from the data (or shifted versions of the data, to ensure serial uniqueness) and with a sufficient number of chips such that "bursty" channel error models can be tolerated. A further constraint on the delimiter choice is that delimiters must not adversely affect the eye diagram of the complete packet. The lack of long strings of contiguous set or reset chips within the 4-Mbit/s delimiters allows this goal to be attained. The delimiters chosen ensure packet robustness at any signal-to-noise ratio, for any length of packet, over random and burst-error models—all without affecting the receiver eye diagram.

Clock Recovery. The UART-style clock recovery of the 2400-to-115,200-bit/s link uses a single signal edge to set the phase of the recovered sampling clock. This inevitably gives rise to phase jitter on the recovered clock and a consequent signal-to-noise ratio penalty. The phase-locked loop used by the 4-Mbit/s link generates a sampling clock with much less jitter because it uses timing information from many signal edges to set the phase of the clock. An analog phase-locked loop could have been used for clock recovery and might have achieved a low phase jitter, but it would have been unable to achieve the rapid phase lock of a digital phase-locked loop. Rapid phase lock is important in a packetized data system, because it determines the length of the training sequence, or preamble, required at the start of every packet to allow the phase-locked loop to lock.

The lock time is dictated by the accuracy with which the nominal frequency of the phase-locked loop's variable oscillator can be set. The nominal frequency of the variable oscillator in an analog phase-locked loop is highly variable, since it is determined by the (usually poor) tolerance of the resistors and capacitors. By contrast, the nominal frequency of the variable oscillator in a digital phase-locked loop can be locked to a crystal reference with a tolerance of less than 100 ppm. Implementations of digital phase-locked loops have the additional advantage that they can be quickly and easily ported between ASIC designs. The architecture of a typical digital phase-locked loop for the 4-Mbit/s link is shown in **Figure 11**.

Figure 11

4-Mbit/s digital phase-locked loop.



The phase detector is a state machine that compares the edges in the received signal (rx_signal) with those of the recovered clock (rx_clock). Rising edges only occur in rx_signal at PPM chip boundaries. Rising edges of rx_clock should occur halfway between chip cell boundaries. If rx_signal is earlier than expected, then the phase detector produces a Down signal, thereby advancing the phase of rx_clock. If rx_signal is later than expected, then the phase detector produces an Up signal.

The three most significant bits of the 8-bit counter set the phase of rx_clock. The five least significant bits ensure that the counter acts as a low-pass filter, since many Up and Down signals are required to change the phase of rx_clock. The three-bit free-running counter and the comparator together act as a variable phase oscillator. All blocks within the phase-locked loop are clocked by the same system clock. The system clock can be either 40, 48, 56 or 64 MHz, the choice being set by the rollover point of the three most-significant bits of the 8- and 3-bit counters (100, 101, 110, or 111). A 40-MHz system clock means that rx_clock should be very granular, with only five possible phase steps within a chip period. The effective number of phase steps is, however, doubled by making use of both the positive and negative edges of the system clock in the phase detector and sampler. The choice of whether to use positive or negative edges can be made by examining the fourth most-significant bit of the 8-bit counter.

The fast lock of the digital phase-locked loop is further aided by using a dual control loop within the digital phase-locked loop. A lock state machine within the phase detector decides whether the digital phase-locked loop is in or out of lock by examining the average deviation of the rx_clock edges from the rx_signal edges. If the digital phase-locked loop is out of lock, then multiple Up or Down pulses are generated for each edge in rx_signal to ensure rapid lock. Once locked, only single Up or Down pulses are generated since multiple pulses would increase phase jitter on rx_clock.

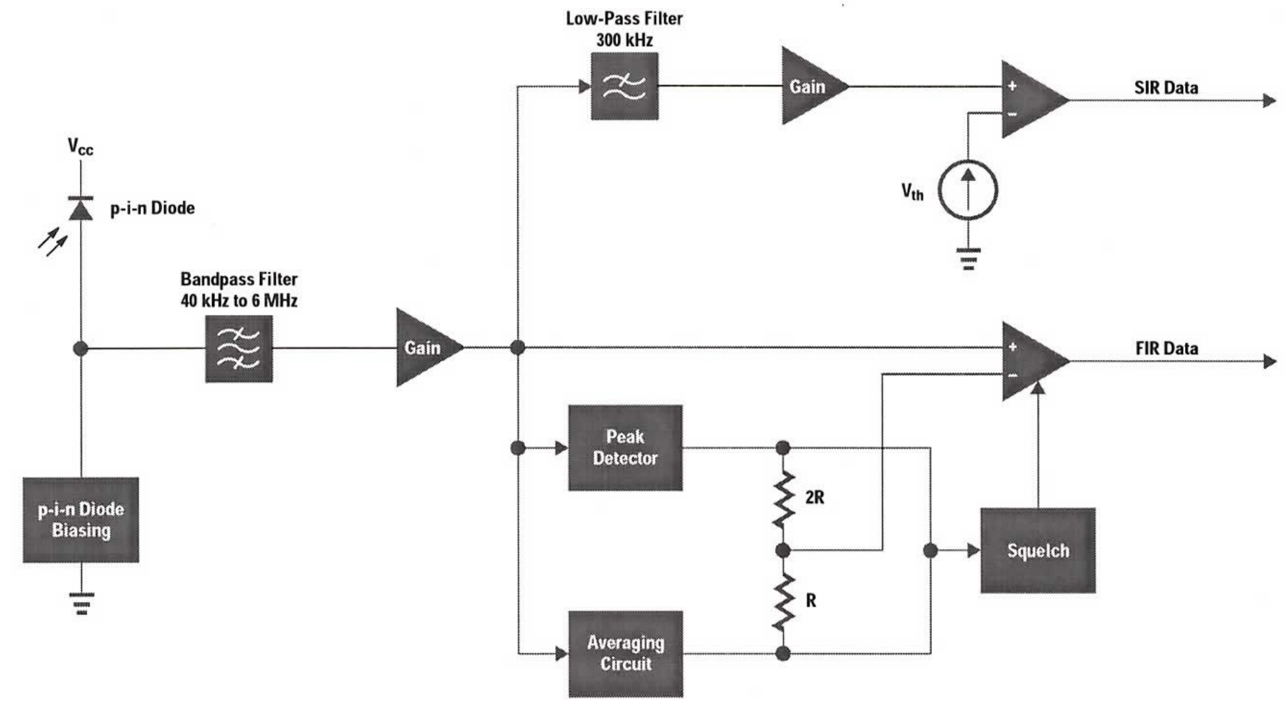
The Hewlett-Packard HSDL-1100 IrDA Transceiver

The HP HDSL-1100 from HP's Communication Semiconductor Solutions Division is the world's first fully IrDA-compliant transceiver capable of operating at all IrDA data rates from 2400 bits/s to 4 Mbits/s. The HSDL-1100 fits within the same small package as its predecessor, the HSDL-1000, which operated at data rates from 2400 bits/s to 115,200 bits/s. The small package size available for pins, IC, passive components, and heat dissipation imposed design constraints on the complexity of the transceiver. The IC uses a low-density bipolar in-house process, which is low in cost and allows quick turn times on wafers for IC development.

Transmitter design was straightforward. However, the multiple data rates, line codes, and large dynamic range made receiver design much more challenging. The receiver's dual-channel architecture is shown in **Figure 12**. A shared p-i-n diode detects all infrared signals with a modulation frequency between 40 kHz and 6 MHz. An amplifier boosts this signal before it is split into separate receiver channels. IrDA signals at 2400 to 115,200 bits/s pass through the serial infrared (SIR) channel*, while 1.152-to-4-Mbit/s signals pass through the fast infrared (FIR) channel. The lower bandwidth of the SIR channel (40 to 300 kHz) means lower noise and allows the SIR channel to meet the IrDA 4 $\mu W/cm^2$ sensitivity requirement. The higher-bandwidth (40 kHz to 6 MHz) FIR channel has higher noise, but still meets the 10 $\mu W/cm^2$ sensitivity requirement for 1.152-to-4-Mbit/s IrDA links. Since the different data rate IrDA links overlap in their modulation spectra, the received signal will appear on both channels. The ENDEC relies on information provided by the protocol to ensure that it listens on the correct channel.

Figure 12

The HDSL-1100 receiver architecture.



* At low rates, such as 2400 or 9600 baud, only the leading edge of the signal passes through the 40-kHz to 6-MHz bandpass filter. The signal is still correctly decoded since the ENDEC is able to tolerate received SIR pulses as short as 1 to 4 μs.

The receiver converts signals from an analog to a digital form by comparing them with a threshold voltage. The two channels have different threshold detection circuits to meet the different requirements for the signals. The SIR channel has a fixed threshold set at the level of the weakest received signals. Although the fixed threshold tends to extend the duration of high-level pulses, the line code for the 2400-to-115,200-bit/s ENDEC is tolerant of pulses that extend to five times their nominal width. The 4-Mbit/s ENDEC is far less tolerant of pulse extension, so a dynamic threshold is required on the FIR channel. The dynamic threshold tracks the 50% level between the peak extensions of the 4PPM signal. A peak detector tracks the 100% level of the signal and an average circuit tracks the 25% level. The 50% threshold level is derived from a 2R-R voltage divider connected to these levels. Between packets, the dynamic threshold drops to zero. This would allow the FIR_Data output to "chatter" on noise or on feedback between the output pin and the p-i-n diode. The 1.152-Mbits/s ENDEC is intolerant of the extra pulses produced by such chatter, so a squelch circuit was added to switch off the FIR_Data output at low signal levels. The dynamic threshold also takes time to settle at the start of a packet, which causes some of the packet's initial infrared pulses to be lost or distorted. While this would be disastrous for the 2400-to-115,200-bit/s link, the 1.152- and 4-Mbit/s packets include a preamble to allow the receiver to settle before decoding data.

Another challenge for receiver design was the dynamic range of infrared signals. IrDA specifications allow received signal strength to vary between 4 $\mu W/cm^2$ and 500 mW/cm^2. This is a dynamic range of 51 dB. Since the p-i-n diode is a square law detector, this dynamic range doubles to 102 dB within the receiver. The receiver achieves this dynamic range by allowing the signal to be clipped while maintaining the timing of the signal. The impedance of the p-i-n diode biasing circuit decreases with signal level, reducing the signal voltage and the receiver amplifier's limit without saturating. The p-i-n diode has also been carefully designed to ensure that the induced signal decays rapidly once an infrared pulse disappears.

The IrDA Protocol Layers

The Infrared Link Access Protocol

IrLAP is the IrDA protocol that provides the basic link layer connection between a pair of IrDA devices. It is based on the HDLC protocol providing functions like connection establishment, data transfer, and flow control.[13,14] However, IrLAP has significant additional features as a result of the specific properties of the infrared medium.

The infrared medium over which IrLAP is required to operate is a point-to-point, half-duplex medium. While the narrow cone angle of IrPHY limits the number of other devices that can be seen, it does increase the probability of hidden devices. In such a situation, one device may see many other devices. However, it does not follow that those devices will see each other. This can result in collisions where transmissions from devices hidden from each other may overlap, resulting in the inability of the receiving device to decode those frames correctly. The characteristics of the infrared medium also result in there being no reliable way to detect transmission collisions. Conventional carrier sensing with collision-detection protocols would therefore be unsuitable, and IrLAP provides a mechanism for ensuring contention-free access to the medium, at least during data transfer.

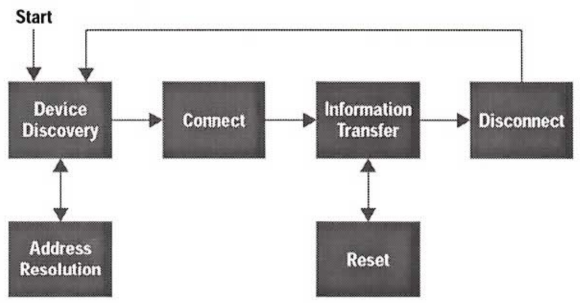
The IrLAP has three distinct phases of operation: link initialization, nonoperational mode, and operational mode. Nonoperational and operational modes are distinguished by the absence or presence of a connection with another device. During link initialization, the IrLAP layer chooses a random 32-bit device address. This address is randomly chosen to negate the need to select and maintain fixed device addresses for all IrDA devices. Although it is unlikely that two or more devices within range of each other will choose the same address, procedures are defined to detect and resolve address collisions. After the link is initialized, the IrLAP layer enters nonoperational mode.

The nonoperational mode is derived from HDLC's normal disconnect mode (NDM). In this mode, all devices contend for the medium. To do this, each device must check that the medium is not busy before transmission. This is achieved by listening for activity—that is, listening for physical layer transitions for at least 500 ms. Transmissions in the normal disconnect mode use link parameters that can be supported by all IrDA devices at a rate of 9600 bits/s. In this mode, the device will initiate device discovery, address resolution (if required), and connection establishment.

Once the connection has been established, the IrLAP layer moves into the operational or, in HDLC terms, normal response mode. This mode is an unbalanced mode of operation in which one device assumes the role of primary station and the other assumes a secondary role. This is the phase in which information is exchanged under control of the primary station. The link parameters are negotiated during the connection setup procedure and remain constant during the connection. During this phase, all other devices within range of either the primary or secondary stations remain idle in the normal disconnect mode. The two communicating devices therefore have unrestricted access to the medium for the duration of the connection. Once the information has been transferred, the link is disconnected and the device returns to the normal disconnect mode. The flow of procedures for the IrLAP layer is shown in **Figure 13**.

Figure 13

The IrLAP procedure flow.



Device Discovery and Address Resolution. The discovery procedure is the process an IrDA device uses to determine whether or not there are any devices within communications range. In doing so, the device discovers the address of any device within range, the version number of the IrLAP protocol operating in each device, and some discovery information specified by the IrLMP layer in each device. The discovery procedure is controlled by the initiating device, which divides the discovery process into equal periods or time slots. The slotted nature of the discover procedure minimizes the likelihood of collisions when there are multiple devices within range.

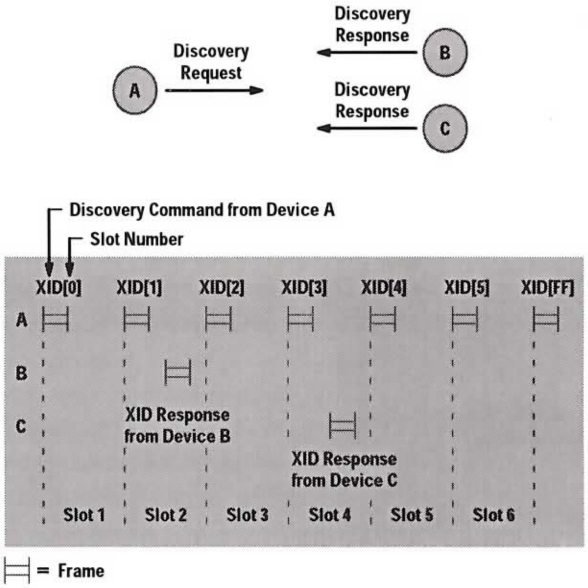
After waiting for a period of 500 ms (normal disconnect mode rules), the initiating device starts the discovery procedure and broadcasts frames marking the beginning of each slot. On hearing the initial discovery slot (which also details the number of slots in the discovery process: 1, 6, 8 or 16), a device randomly selects one of the slots in which it will respond. When the device receives the frame marking its chosen slot, it transmits a discovery response frame. All frames in the discovery procedure use the HDLC unnumbered format of type XID (exchange identification).* An example of the discovery process is shown in **Figure 14**.

Figure 14 shows a three-device scenario in which device A is within range of devices B and C. Device A initiates the discovery process by transmitting a discovery XID command frame which, in this case, indicates that this is a six-slot discovery process and that this is the initial slot. Device A continues to transmit discovery command XID frames indicating the appropriate slot number. The final frame, after slot 6, is indicated by a slot number 0xFF. The final slot also contains information about the initiating device.

* In this context XID is a type of HDLC frame as specified in the ISO standard.

Figure 14

The discovery procedure.



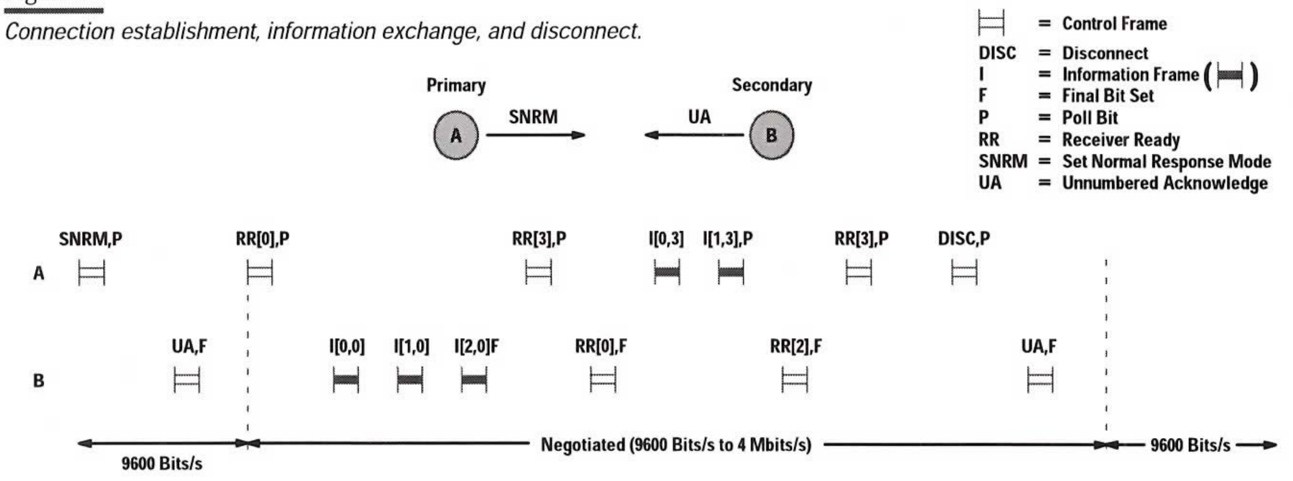
When the initial discovery XID command frame is received, devices B and C randomly choose slots in which to respond—in this example, slots 2 and 4. Device B then waits until it hears the discovery XID command indicating slot 2, and responds with a discovery XID response frame containing information about itself. Similarly, device C transmits a response during slot 4. Once the discovery process is over, all devices have the address and other information of all the devices within range: that is, device A has information about devices B and C, while devices B and C each have knowledge of device A. However, devices B and C are mutually hidden and as a result have no information about each other. This discovery information is passed to the upper layers whose responsibility it is to determine if there are any address collisions that need to be dealt with.

Should any of the devices that participated in the discovery process have duplicate addresses, then an address resolution process can be initiated. Address resolution follows a procedure similar to the discovery process, except that the device detecting the address conflict initiates the procedure, and resolution involves only the devices that have conflicting addresses. In this case, the initiating device transmits an address resolution XID command directed at the conflicting address. Devices with this address select another random address and a slot in which to respond. The initiator transmits the slot markers as before, and the previously conflicting devices respond in the appropriate slot. Once the process is over, each device should have a unique address. In the unlikely event that an address conflict still exists, the procedure can be repeated.

Connection Establishment. Once the discovery and address resolution processes are complete, the application layer may decide that it wishes to connect to one of the discovered devices. To connect, the application layer will issue a connection request which will ultimately result in the appropriate IrLAP service primitive being invoked. The IrLAP layer connects to the remote device by transmitting a set normal response mode (SNRM) command frame with the poll bit set. This command informs the remote device that the source wishes to initiate the connection and the poll bit indicates that a response is required. Assuming the remote device can accept the connection, it responds with an unnumbered acknowledge (UA) response frame with the final bit set. This indicates that the connection has been accepted. Under normal circumstances, the device that initiates the connection (transmits the set normal response mode) will become the master, or primary, device, and the other device will become the slave, or secondary device. An example of connection establishment is shown in **Figure 15**.The notation used in the frames in **Figure 15** has the general form I(x,y) and RR(y), where x is the sequence number of the information frame and y is the sequence number of the next frame the source device expects to receive from the destination device.

Figure 15

Connection establishment, information exchange, and disconnect.



The connection establishment takes place in normal disconnect mode (9600 bits/s), and once this is completed, the two devices will be in normal response mode. While in normal response mode, the devices can exchange data at any IrDA defined rate. However, not all IrDA devices will support all IrDA data rates or link parameters. It is therefore necessary for the devices to negotiate the parameters for normal response mode during connection setup. IrDA has defined several link parameters that can be negotiated:

- Data rate
- Maximum turnaround time
- Data size
- Window size
- Number of additional start of frame symbols (BOFs)
- Minimum turnaround time
- Link disconnect threshold time.

Data rate defines the data transfer rate during normal response mode (9600 bits/s to 4 Mbits/s), while maximum turnaround time defines the length of time either device may transmit before giving the other device a chance to transmit (50, 100, 250, or 500 ms). Data size determines the maximum length of the data field in an information frame (64 to 2048 bytes), and, in combination with the retransmission window size, which defines the number of outstanding frames that may be unacknowledged, allows devices with only limited resources to restrict the rate at which they will receive data. Number of additional BOFs and minimum turnaround time relate to physical layer restrictions, while link disconnect threshold time determines how long a device will wait without receiving a response from another device before assuming the link has failed and informing the upper layer that the link has disconnected.

Well-defined rules exist that ensure that after the set normal response mode-UA exchange has been completed, both devices will know the negotiated normal response mode parameters. Once both devices are in normal response mode, the primary device polls the secondary device by transmitting a receiver ready (RR) frame with the poll bit set, thereby initiating the information exchange phase.

Information Exchange and Link Reset. The information exchange procedure operates in a master-slave mode in which the primary device controls the secondary device's access to the medium. The primary device issues command frames to the secondary device which responds with response frames. To ensure that only one device can transmit frames at any one time, a permission-to-transmit token is exchanged between the primary and secondary devices. The primary device passes the permission-to-transmit token to the secondary by sending a command frame with the poll bit set. The secondary device returns the token by transmitting a response frame with the final bit set. The secondary device can only retain the token while it is transmitting data, and it must return it to the primary device if it has no data to transmit or if it reaches the maximum turnaround time. The primary device, however, within the limits imposed by the maximum turnaround time, can hold the token even if it has no data to transmit.

Although the physical layer has been designed to provide a low bit error rate channel, the dynamic nature of the infrared connection results in a possibility that frames may be lost in transit because of corruption by noise. To cope with this, the IrLAP protocol uses a sequenced information exchange scheme with acknowledgments. Should a frame be corrupted by noise, the CRC will highlight this error and the frame will be discarded. At the IrLAP layer, this error will be detected by virtue of the noncontiguous sequence numbers on the information frames. The IrLAP protocol implements an automatic repeat request strategy in the same manner as HDLC with options of using stop and wait, go back to N, and selective reject retransmission schemes.[13] This strategy allows the IrLAP layer to provide an error-free, reliable link to the IrLMP layer. An example of an error-free information exchange between two devices is shown in **Figure 15**.

Under exceptional circumstances, however, it may not be possible for the IrLAP entities in each device to recover from an error condition while maintaining the sequenced delivery of error-free information (I) frames. In this case, the IrLAP entity is allowed to reset the link. This reset involves discarding any undelivered information and reinitializing the sequence numbers and timers for the link. Although this may result in the loss of data, which the higher-level layer must deal with, it does allow the link to recover without the need for a total disconnection.

Connection Termination. Once the data exchange has taken place, the IrLAP link may be disconnected by either the primary or secondary devices. Should the primary wish to disconnect, it sends a disconnect command to the secondary device with the poll bit set. The secondary responds by returning an unnumbered acknowledge frame with the final bit set. Both devices will now be in normal disconnect mode, and the default normal disconnect mode parameters (9600 bits/s data rate) will apply. If the secondary wishes to disconnect, it transmits a request disconnect response with the final bit set when it is polled by the primary. The primary will then respond by transmitting a disconnect command, and both devices will be in normal disconnect mode. An example of a primary-initiated disconnection is shown in **Figure 15**. Once the two devices are in normal disconnect mode, the medium is free for any other device to initiate the discovery, address resolution, or connection procedures.

The Infrared Link Management Protocol

The Link Management Protocol (IrLMP) is layered on top of IrLAP, and has two main functions: application and service discovery and multiplexing of application level connections over the single IrLAP connections. The IrLMP layer allows individual service users (applications) to connect and exchange information with similar entities in the peer device, independent of any other service users that may be using the IrLAP connection. The IrLMP layer provides multiple independent channels to the IrLMP layer in the remote device. The IrLMP layer also provides a service with which applications can locally register themselves and some significant parameters in an information base. Services are also provided that enable those applications to access equivalent information in the information base of remote devices. Using this service, an application does not need prior knowledge of the applications in a remote device. This is an extremely useful feature for the kind of ad-hoc interactions typical of IrDA devices.

The two main functions provided by IrLMP are split between two sublayers. The Link Management Multiplexer (LM-MUX) provides the facilities for multiplexing application level connections over a an IrLAP connection between a pair of devices. The Link Management Information Access Service (LM-IAS) provides the services necessary to allow applications to discover devices and access the information in the information base of a remote device.

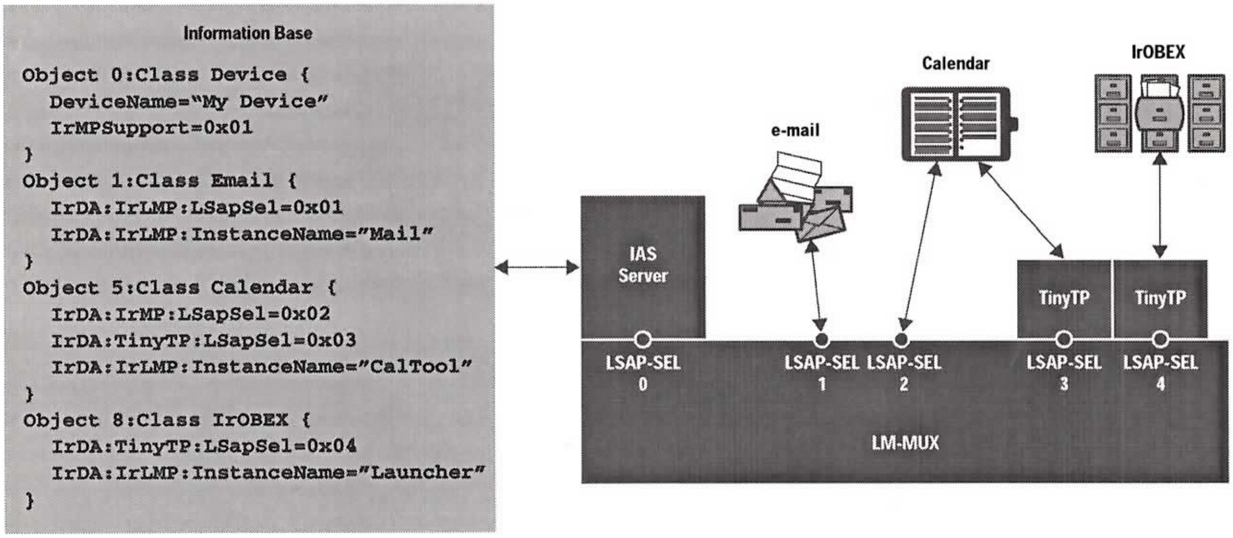
The Link Management Multiplexer. The LM-MUX adds two bytes of overhead to the IrLAP information frame, which are primarily used for addressing the individual multiplexed connections. The address fields uniquely identify the link service access points (LSAPs) in both the source and destination devices. Each LSAP is addressed by a seven-bit selector (LSAP-SEL), and LSAP-SELs within the range 0x01 to 0x6F can be used by applications. LSAP-SELs 0x00 and 0x70 are reserved for the information access service server and the connectionless data service respectively. The remaining LSAP-SEL values, 0x71 to 0x7F, are reserved for future use. Connections between IrLMP service users are called LSAP connections, and although an LSAP may terminate other LSAP connections, there is only one LSAP connection between any pair of LSAPs. All LSAP connections use the single IrLAP connection between the pair of devices.

Information Access Service. The information access service maintains information about the services provided by the host device and provides services that allow access to the information base on remote devices. The information access service allows devices to discover which services are available on the host device and provides the configuration information necessary to access those services. As an example, the most common piece of information required is the LSAP-SEL value, which tells where a particular service is located.

The information stored in the information base consists of a number of objects. Each object belongs to a specific class, and there may be several objects of the same class in the information base. The class defines the attributes that are present in each object, and these attributes can be assigned a particular value. The attributes of a class can be of type user string, octet sequence, signed integer, or missing. **Figure 16** shows an example of the information access service database for a device offering three unique services.

Figure 16

Example information access service database.



Information Base

```
Object 0:Class Device {
  DeviceName="My Device"
  IrMPSupport=0x01
}
Object 1:Class Email {
  IrDA:IrLMP:LSapSel=0x01
  IrDA:IrLMP:InstanceName="Mail"
}
Object 5:Class Calendar {
  IrDA:IrMP:LSapSel=0x02
  IrDA:TinyTP:LSapSel=0x03
  IrDA:IrLMP:InstanceName="CalTool"
}
Object 8:Class IrOBEX {
  IrDA:TinyTP:LSapSel=0x04
  IrDA:IrLMP:InstanceName="Launcher"
}
```

The example shows a device with three individual applications: e-mail, calendar, and IrOBEX (file transfer application). The information base contains three objects associated with these applications. The required Object 0 is always present within the information access service database, and it provides information about the device name and the version of IrLMP the device supports. All other devices can address Object 0 to get this information. Objects in the information base typically detail information about the services provided—for example, the LSAP-SEL where these services can be accessed. In the case of the calendar application, this service can be accessed using the Tiny TP flow-control mechanism on LSAP-SEL 3, or directly on LSAP-SEL 2. The difference is encoded in the attribute name.

The IrLMP layer provides several service primitives to access information access service data. However, the only mandatory service is GetValueByClass. This service requires the service user to provide the class and attribute names of the service it is interested in. The service returns a list of object identifiers and attribute values for all objects in the information base with the requested class and attribute name. Referring to the example in **Figure 16**, if a peer device issued a GetValueByClass with parameter Calendar for the class name and IrDA:IrLMP:LSapSel for the attribute name, the service would return a single element list with the entry containing object identifier 5 and attribute value 2.

Tiny TP Flow Control Mechanism

Although the IrLAP layer does have provisions for flow control, its use can result in deadlock situations, particularly where more than one IrLMP connection is operating. Such a deadlock situation can occur if an application in one device is waiting for its peer application to send it some data before releasing its buffer space. However, another connection may use up the remaining buffer space, causing the IrLAP layer to flow-control the link until buffer space becomes available. If both connections are waiting for data from the remote device before freeing the buffers, then clearly a deadlock has occurred that cannot be resolved without some form of higher-level intervention such as a system reset.

To overcome this problem, IrDA provides the lightweight transport protocol called Tiny TP.[8] Tiny TP adds a single byte of overhead to each frame and provides a per-LSAP-connection credit-based flow control mechanism with the possible segmentation and reassembly of service data units of up to 4 Gbytes in size. When a Tiny TP connection is initiated, the maximum service data unit size is negotiated and some initial credit is extended to each connection endpoint. Sending data causes the credit to be decreased by one, and periodically the receiver issues more credit. Without credit, the transmitter cannot send any data. It must wait until such times as the receiver extends it some more credit. Using Tiny TP, a device can ensure that credit is distributed among its applications, ensuring that the applications can communicate without reducing the buffer space to such a degree that IrLAP flow control must be used.

Conclusion

IrDA has completed the core standards necessary to enable any mobile computing platform with ad-hoc, point-and-shoot infrared communications from 2400 bits/s to 4 Mbits/s. Support for the IrDA platform from a wide variety of manufacturers is now becoming apparent, as many products—ranging from printers to laptop PCs and PDAs to mobile phones—are being released with IrDA capability. All these devices will have the ability to interoperate with one another should that be required. With over 130 companies actively maintaining membership in IrDA, currently released IrDA-enabled products represent only the tip of the iceberg. In the coming months and years, it is expected that more and more computing and other devices will be released with built-in IrDA capability.

However, providing the hardware platform to support IrDA is only half the story. Current activity within IrDA is directed at finishing off the IrDA series of standards to enable application-level developers to access the IrDA features in a uniform and efficient manner. The needs of legacy serial/parallel applications have been addressed with the IrCOMM standard. Legacy networking applications will be able to use the IrDA features implemented in the forthcoming IrLAN protocol. However, it is expected that a new class of applications will be developed with the express purpose of using the unique features of IrDA-enabled devices. The IrOBEX protocol, when completed, will provide application programmers with a generic method by which data can be exchanged with other applications without having to know the details of the destination application. As an example, transferring a graphic to another PDA (which will display it) or to a printer (which will print it) will be no different from the source application's point of view. Alternately, a more flexible approach to accessing the IrDA communications facilities will be to directly access them through the operating system's application programming interface. An example of this is the WinSock-style API to IrDA, called IrSock,[15] currently being developed for the Microsoft® Windows 95 operating system.

In conclusion, the future for infrared is bright. With cross-industry support, IrDA is fast becoming the ubiquitous infrared communications system for portable and peripheral devices. Although legacy support for other infrared systems will persist for some time to come, the IrDA standard is now used on so many platforms that it is unlikely any new systems will be anything other than IrDA-enabled.

References

1. *IrDA Marketing Requirements—Basis for the IrDA Technical Standards*, Version 3.2, The Infrared Data Association, November 23, 1993.

2. *Serial Infrared (SIR) Physical Layer Link Specification*, Version 1.0, The Infrared Data Association, April 27, 1994.

3. *Serial Infrared Link Access Protocol (IrLAP)*, Version 1.0, The Infrared Data Association, June 23, 1994.

4. *Link Management Protocol (IrLMP)*, Version 1.0, The Infrared Data Association, August 12, 1994.

5. *Serial Infrared Physical Layer Link Specification*, Version 1.1, The Infrared Data Association, October 17, 1995.

6. *Link Management Protocol (IrLMP)*, Version 1.1, The Infrared Data Association, January 23, 1996.

7. *Serial Infrared Link Access Protocol (IrLAP)*, Version 1.1, The Infrared Data Association, June 16, 1996.

8. *Tiny TP: A Flow-Control Mechanism for use with IrLMP*, Version 1.0, The Infrared Data Association, October 25, 1995.

9. *IrCOMM: Serial and Parallel Port Emulation over IR (Wire Replacement)*, Version 1.0, The Infrared Data Association, November 7, 1995.

10. *LAN Access Extensions for Link Management Protocol (IrLAN)*, Proposal, Version 1.0, The Infrared Data Association, January 1, 1996.

11. *Object Exchange Protocol (IrOBEX)*, Draft, Version 0.5f, The Infrared Data Association, July 24, 1996.

12. S. L. Harper and R. S. Worsley, "The HP 48SX Calculator Input/Output System, *Hewlett-Packard Journal*, Vol. 42, no. 3, June 1991.

13. *Information technology—Telecommunications and information exchange between systems—High-level data link control (HDLC) procedures—Elements of procedures*, ISO/IEC 4335, International Organization for Standardization, December 15, 1993.

14. *Information technology—Telecommunications and information exchange between systems—High-level data link control (HDLC) procedures—Classes of procedures*, ISO/IEC 7809, International Organization for Standardization, December 15, 1993.

15. *Infrared Sockets (IrSockets) Specification: Functional Specification*, Revision 1.0, The Microsoft Corporation, July 10, 1996.

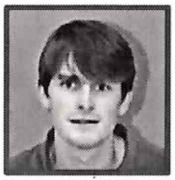
Microsoft is a U.S. registered trademark of Microsoft Corporation.

Iain Millar

As a member of the technical staff at HP Laboratories in Bristol, England, Iain Millar is involved in the development of protocols for the next generation of IrDA systems. He attended the University of Aberdeen in Scotland where he received a BSc (Eng.) degree (1988) in electrical and electronic engineering and a PhD (1995) in the area of fault tolerant protocols for LANs.

Martin Beale

Martin Beale is a member of the technical staff at Hewlett-Packard Laboratories in Bristol, England. He is working on the physical layer for the next generation IrDA systems. He earned a PhD degree (1994) in reduced complexity decoding of convolutional codes from the University of Cambridge. Outside of work he enjoys rock climbing, walking, cycling, skiing.



Bryan J. Donoghue

Bryan Donoghue is a member of the technical staff at HP Laboratories in Palo Alto where he is working on the digital system design for high-speed wireless radio LANs. He received a MEng degree (1991) in electrical and electronic engineering from Loughborough University in England. Bryan was born in Llanelli, Wales and outside work he enjoys traveling, learning foreign languages, backpacking, and cycling.

Kirk W. Lindstrom

A member of the technical staff at HP's Communication Semiconductor Solutions Division, Kirk Lindstrom is responsible for the design of ICs for infrared products. He joined HP in 1978 and since that time he has worked on the design of optocouplers, fiber-optic modules, and infrared transceivers. He holds a BS degree in EECS (1979) from the University of California at Berkeley. Besides being an ardent windsurfer, he also has an interest in doing and writing about investing.

Stuart Williams

A project manager at HP Laboratories in Bristol, England, Stuart Williams is responsible for the infrared communications group. He has worked on various infrared protocol-related projects since he joined HP in 1992. He has a PhD degree (1986) from the University of Bath. Stuart was born in Rugby, Warwickshire, England, is married and has two sons. Biking and sailing occupy his free time.

▶ Go to Next Article

▶ Go to Journal Home Page

An Introduction to IrDA Control

Robert Stuart, IrDA Product Manager

Presented at Portable Design East on August 31, 1998

This paper is an overview of the IrDA* Control system technology. The areas covered will be the Physical Layer (PHY), Media Access Control Layer (MAC) and the Logical Link Control Bridge Layer (LLC).

NOTE: *The Infrared Data Association: IrDA was established in 1993 to set and support hardware and software standards, which create infrared communications links. IrDA standards support a broad range of computing, communications, and consumer devices. International in scope, IrDA is a non-profit corporation head quartered in Walnut Creek, California, and led by a Board of Directors, which represents a voting membership of more than 160 corporate members worldwide. As a leading high technology standards association, IrDA is committed to developing and promoting infrared standards for the hardware, software, systems, components, peripherals, communications, and consumer markets.

IrDA can be contacted at: PO Box 3883, Walnut Creek, CA, 94598; Web: www.irda.org, Phone: 925-943-6546, Fax: 925-943-5600.

INTRODUCTION

IrDA Control technology differs from the classical IrDA Data technology in several key characteristics. IrDA Data is a peer-to-peer file-oriented data transmission system. The link range was specifically designed for a one-meter range to meet a variety of requirements.

IrDA Control is a command and control architecture for communication with wireless peripheral devices such as mice, keyboards, gamepads and joysticks. This system is specifically oriented towards control data packets, and is not intended to pass files. The purpose is to pass short control packets between the host device and the remote input devices.

SYSTEM OVERVIEW

The IrDA Control system is a polled-host topology. The host device polls up to eight peripheral devices in an ordered sequence, providing service requests and handling the peripheral device responses.

The host may be a Personal Computer (PC) with peripheral devices such as a mouse and keyboard. Once the system boots up, the remote (wireless) keyboard and mouse will operate with the host PC in the same manner as a wired keyboard and mouse. The PC system drivers acknowledge the wireless mouse and keyboard, and they will work in addition to the normal mouse and keyboard, if desired.

When the peripheral devices are brought into operation, the system performs an enumeration sequence so that the host knows the peripheral device, what type of device it is and how it is expected to act. Once enumerated successfully, the device will then be bound to the host when it is to be used. Up to eight peripheral devices may be held in the device enumeration list and up to four devices actively bound and communicating with the host at one time.

If a mouse is operating, and then is not used for a few seconds, the binding will be dropped and the enumeration still held. When the mouse is again used, the system will rebind it and accept inputs from it. If the mouse remains idle and another idle device needs service, if it has previously been enumerated, it will be bound and service will be provided as long as overall system requirements are not exceeded.

The IrDA Control system has an operating range of about seven meters, on average. Peripheral devices may be used in a short-range environment, or at longer ranges such as sitting on the couch in a family room at home.

The use of IrDA Control is not limited to the PC environment. It will work as effectively with Set Top Boxes and other consumer devices and will lead to new interactive remote devices for use with these products.

The system layers covered in this paper are shown in Figure 1.

The applications and access layer software reside on top of the physical layer. These will be described from the bottom up.

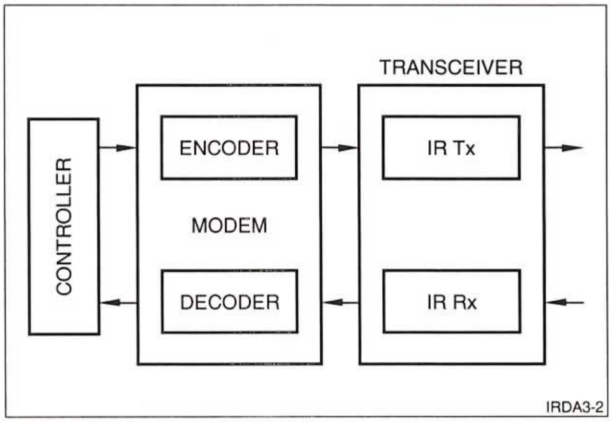


Figure 1. System Layers

PHYSICAL LAYER (PHY)

The IrDA Control system uses a PHY that is different from the earlier data-oriented IrDA 1.0 and 1.1 standards. IrDA Control uses a 16 Pulse Sequence Modulation (16 PSM) format. Each data bit encapsulates a 1.5 MHz subcarrier frequency. The overall payload capability for the system is 75 kbps*.

NOTE: *Refer to the complete IrDA Control 1.0 specification, Copyright Infrared Data Association.

The IrDA Control specification defines the transmission speeds, modulation schemes, infrared wavelengths of the optical signals emitted by the transmitter and those signals received by the receiver.

The specification does not mandate the actual signals in the encoder/decoder process or the internal signals in the IR transceiver.

The data transmission process is handled by optical transceiver devices that incorporate both the transmission Light Emitting Diode (LED) and the Photodetector (PD) circuits and amplifiers.

An encoder and decoder reside in the bit-stream path and handle data coding and the modulation process. Data is passed from the controlling device to the encoder/decoder and then on the transceiver. The 1.5 MHz subcarrier process and the coding of the transmission symbols were chosen to minimize possible interference with other transmission systems.

The basic flow of information shown in Figure 2 works for both the host and peripheral side of the system. In both cases, when actions are to be completed, the controller makes a decision and sends data out through the infrared link to the other device.

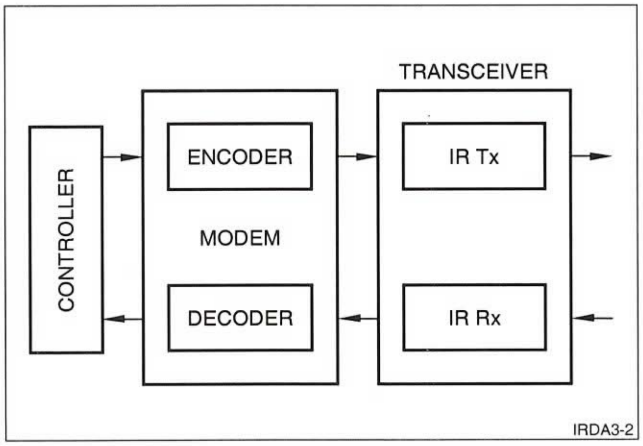


Figure 2. PHY Layer Block Diagram

In the case of a mouse, position or button press information is held in the microcontroller. When polled by the host, the mouse will respond, informing the host that it has information to send. The host will then request the information and the mouse will send it. The mouse controller will pass the data to the Modem function, which will handle all coding and modulation details. The transceiver performs the electrical to optical translation between systems.

The encoder automatically formats the data stream in the 16 PSM scheme for transmission.

A time defined as 'symbol time (Dt)' is equally divided into eight slots defined as 'chips', and a pulse is allowed only during two or four of those chip periods. Each chip time (Ct) is given by the following equation:

$Ct = Dt \div 8$

Information is transmitted according to the pulse pattern of the sequence. Unique sets of four bits correspond to a specific symbol value. The Data Bit Sets of the 16 PSM symbols are shown in Table 1. The waveforms that have legal pulse sequences are defined as 16PSM Data Symbols, or simply as Symbols.

In the 16 PSM scheme, four bits of information can be transmitted within a single symbol time. Accordingly, there are 16 waveforms defined as 16 PSM Data Symbols. Each unique set of four bits corresponds to one of 16 symbol values, and is defined as a Data Bit Set (DBS).

Table 1. 16 PSM Data Symbol Representation

DATA VALUE (HEX)	DATA BIT SET (DBS)	16 PSM DATA SYMBOL
0x0	0 0 0 0	1 0 1 0 0 0 0 0
0x1	0 0 0 1	0 1 0 1 0 0 0 0
0x2	0 0 1 0	0 0 1 0 1 0 0 0
0x3	0 0 1 1	0 0 0 1 0 1 0 0
0x4	0 1 0 0	0 0 0 0 1 0 1 0
0x5	0 1 0 1	0 0 0 0 0 1 0 1
0x6	0 1 1 0	1 0 0 0 0 0 1 0
0x7	0 1 1 1	0 1 0 0 0 0 0 1
0x8	1 0 0 0	1 1 1 1 0 0 0 0
0x9	1 0 0 1	0 1 1 1 1 0 0 0
0xA	1 0 1 0	0 0 1 1 1 1 0 0
0xB	1 0 1 1	0 0 0 1 1 1 1 0
0xC	1 1 0 0	0 0 0 0 1 1 1 1
0xD	1 1 0 1	1 0 0 0 0 1 1 1
0xE	1 1 1 0	1 0 1 0 0 1 0 1
0xF	1 1 1 1	1 1 1 0 0 0 0 1

The encoder will place each bit of the DBS into one of the eight chip times in each symbol, as explained above. The encoder will also insert the 1.5 MHz subcarrier into each bit envelope in preparation for transmission of the symbol.

For the Data Bit Set corresponding to the hex value in the left column, the encoder will format the 16 PSM data shown on the right. This data symbol will then be transmitted over the optical link.

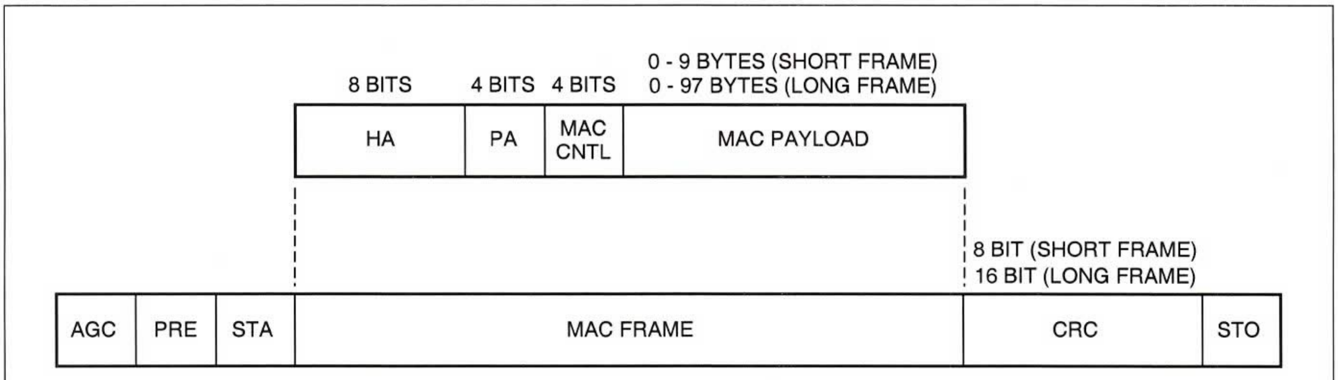
The system packet structure is shown in Figure 3. Two types of packets are used in the IrDA Control system: short packets and long packets.

Each packet consists of six fields: The Automatic Gain Control (AGC); Preamble (PRE); Start Flag(STA or STL); MAC frame; Cyclic Redundancy Check (CRC, either CRC-8 or CRC-16); and Stop Flag (STO).

Data transmission starts with the leftmost bit in each field. The AGC field is used to set the AGC level in the receiver.

The Preamble is used to attain clock synchronization. The Start Flag is used for symbol synchronization. The MAC frame is passed, the CRC is sent and the Stop Flag sent to end the transmission.

The 1.5 MHz subcarrier pulses in the data bits are transmitted for a logical '1' and are not transmitted for a logical '0'.



NOTES:
HA: Host Address field
PA: Peripheral Address field
MAC CNTL: MAC Control field

IRDA3-3

Figure 3. MAC Frame Structure

The diagram in Figure 4 shows an example of data value 0xF5 is being sent out. The lower half of the data byte is sent first, so the value F5 is actually transmitted as 5F.

The Subcarrier Emission Pulse Chip (SEPC) is the string of ten 1.5 MHz subcarrier pulses that are inserted into each logical '1' data bit of the eight chip times in the transmitted 16 PSM data Symbol.

When the data stream is detected at the receiver, the 1.5 MHz subcarrier pulses will be extracted and the coded Data Symbol will be forwarded to the decoder. The Data Bit Set associated with each 16 PSM Data Symbol will be extracted and passed to the MAC layer. The system will recognize the control symbols such as Start and Stop and translate them appropriately.

The system specifications call for a minimum five-meter operating range. Various distances and operating angles between host and peripherals are described in the full specification. The basic operating environment calls for a ±30° angle from the transceiver in the horizontal plane and ±15° in the vertical plane.

Detailed specifics of the ranges, angles and operating conditions are in the complete IrDA Control specification.*

NOTE: *Refer to the complete IrDA Control 1.0 specification, Copyright Infrared Data Association.

MEDIA ACCESS CONTROL LAYER (MAC)

The IrDA Control system consists of hosts and peripherals between which infrared communication takes place. The host manages its communications with multiple peripherals on a time division basis, using polled-response handshakes.

The host polls all of the bound peripherals to determine which items need to be serviced. The peripherals respond to the poll from the host, and do not initiate transmission. The peripheral devices do not transmit unless they are given response permission.

The only exception is when the host is asleep and a peripheral initiates a wakeup call for service. Then the host steps back into the polling sequence and looks for devices to service. If there is no transmission between the host and any peripheral for a set time, then the host will again enter sleep mode.

Generally, hosts do not communicate with each other, however there could be times when they need to do so, if there are multiple hosts in a room. Usually if multiple hosts are present, they detect each other and dither their transmissions to reduce the possibility for interference.

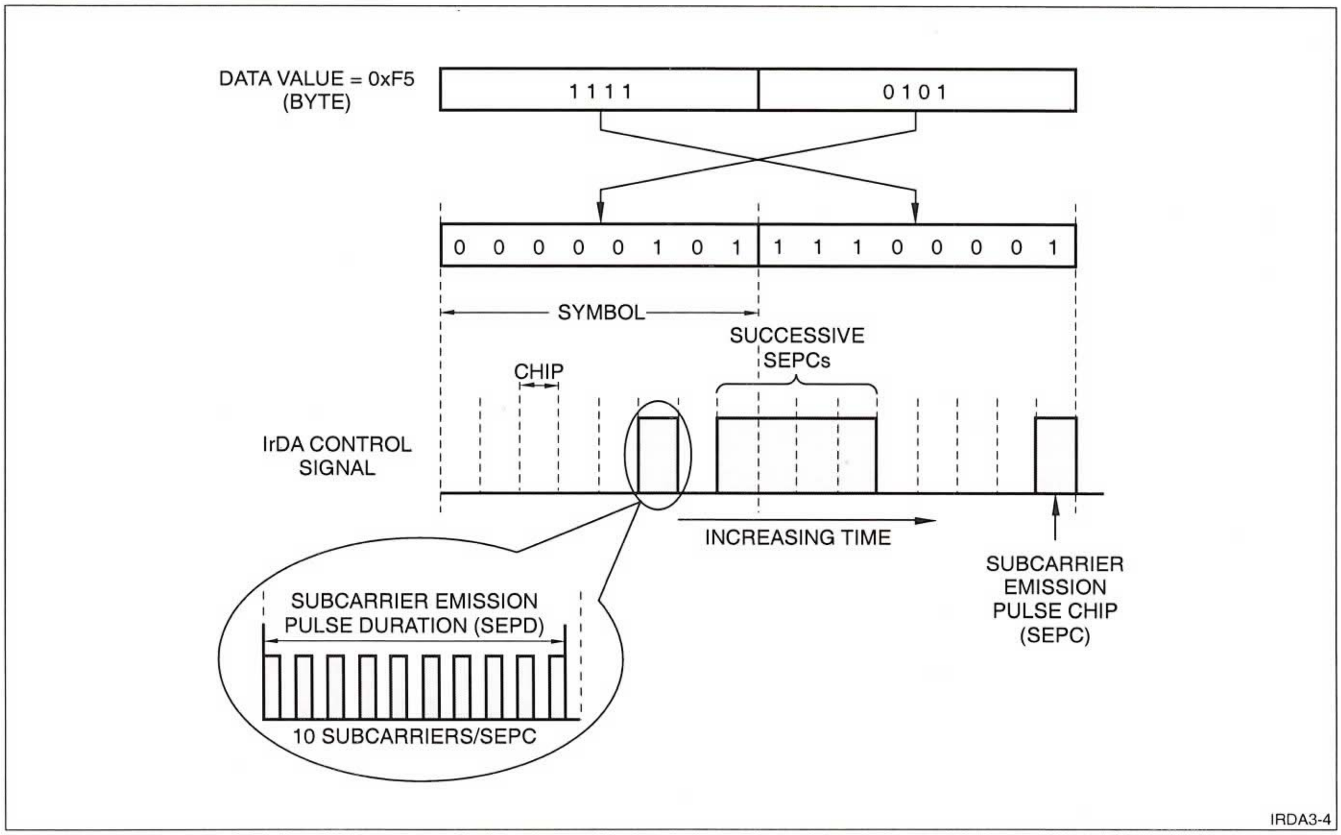


Figure 4. Data Transmission Coding Example

Each device has an address and identifier that clearly identifies hosts and peripherals. An 8-bit host address (HADD) and a 16-bit host ID (HostID) identify a host. A host address may be set at the factory, or be determined while the host is set up.

A peripheral is identified by a 32-bit physical ID (PFID). A host and a peripheral have to exchange address/ID information as part of a process called enumeration.

A logical 4-bit peripheral address (PADD) is uniquely assigned to each peripheral by the host to establish 'active' communication. This procedure is a part of a process called binding, which is performed when an enumerated peripheral requests communication with the host. The ID numbers are used only in the beginning of a communication to identify the devices. After the identification, hosts/peripherals are identified only by their address.

The requirements for IrDA Control communication vary depending on the application. In order to comply with various application requirements, three operational modes are offered for a host.

Mode-0: Sleep Mode

This is a 'Low resource usage' mode to minimize power consumption when a host and its peripherals do not need to communicate. This is also the default mode for each host.

Mode-1: Normal Mode

This is the normal operational mode of the host. This mode supports peripherals that may have different bandwidth requirements. Peripherals supported include devices that must be handled within certain time limits [Critical Latency peripheral (CL)], like joysticks and game pads.

Peripherals that normally do not have critical latency requirements [Non-critical Latency peripheral (NCL)], like Remote Control units are also supported. Keyboards and mice could be handled as NCL or CL peripherals under this mode. A CL peripheral is able to support CL polling rate.

An NCL peripheral is not able to support CL polling rate and is always polled at the NCL polling rate. A host must guarantee that a CL peripheral is polled every 13.8 ms.

Mode-2: IrDA-coexistence mode

This operating mode is available to allow coexistence of IrDA SIR version 1.1 data communication and IrDA Control communication.*

NOTE: *Refer to the complete IrDA Data specification version 1.1 or 1.2 for detailed information on SIR and FIR modes of operation.

The host may move between any of the three modes listed above. It is not required that all hosts support all three modes.

When enumerated, the peripheral identifies the type of service that it requires, so that the host knows whether CL or NCL support is to be used. Critical Latency devices have priority over Non Critical Latency devices, which may not be serviced within the 13.8 ms cycle, depending on system resources and how they are being used. In some cases they may unbind and wait for a service slot.

If four CL devices such as joysticks are actively engaged in a game, then the NCL devices may not be poled for an extended period of time. They do remain enumerated and known to the host. Should the play of the game slow such that the CL activity decreases then the NCL devices can rebind and be serviced. An example of this is to stop the play and enter some text or player names or other similar activity.

FRAMES

Two types of MAC frames are defined based on the maximum MAC payload data length that can be transmitted by a host or a peripheral. One is a short frame and the other is a long frame. A short frame can accommodate up to 9 bytes of MAC payload data and must be transmitted with the STS flag, STO flag and CRC-8. These are shown in Figure 5.

A long frame can accommodate up to 97 bytes of MAC payload data and must be transmitted with the STL flag, STO flag and CRC-16. Long frames are suitable for larger data exchanges.

Host devices and peripheral devices may always use short frames. Host devices may use long frames in Mode-1 only. Peripheral devices may use long frames only when responding to a polling packet from a host device whose long frame enable bit is set to '1', which occurs when the host is in Mode-1. A host device and a peripheral device are prohibited from both using a long frame in the same polling procedure (in the polling frame from a host as well as the responding frame from a peripheral).

In this case it is also possible that the NCL polling cycle may be stretched if several NCL devices are exchanging long frames. Once the activity is finished, the normal polling cycle will be resumed.

The basic polling cycle for the IrDA Control system is defined as 13.8 ms. Up to four CL peripherals can be polled with short frames within this cycle time. The basic polling cycle time is dependent on the minimum interval between inputs from a peripheral input device, such as a joystick or gamepad. These devices have the most critical response time. Keyboards and mice are more flexible with regards to actual response time.

A Non Critical Latency device is not guaranteed a poll within the 13.8 ms time. The entire polling cycle time is defined as the time period in which all bound peripherals can be polled by a host. The host has to manage all of the peripherals so that the entire polling cycle time does not exceed 69 ms.

The possibility exists that cases may arise when the cycle time is shorter than the time required servicing all of the items in the list. The host will try to service all of the devices on the next poll cycle. A peripheral device that misses one or two poll cycles will not immediately be unbound. The Peripheral must not acknowledge approximately 100 polls before the host drops the binding.

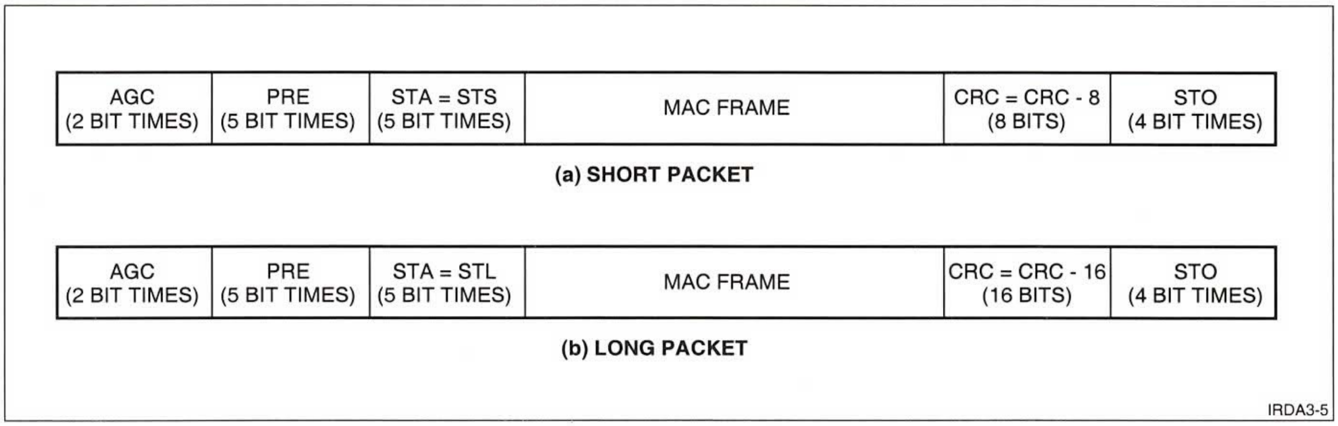
With this information in mind, long frames are only applicable for transmission when CL devices are not bound on the system, or their service requirements do not restrict the system from servicing long frames.

The MAC frame field structure is shown in Figure 3. The Host Address and Peripheral Address fields and the number of bits associated with each are shown.

The MAC control field has a variety of functions. It is used to communicate packet direction, bind timer restarted, long frame enable, device hailing and polling requests.

Enumeration is the procedure in which a host and a peripheral recognize (discover) each other to enable communication between them. The host identifies the peripheral using the peripheral physical identifier (PFID) and the peripheral identifies the host using a host address. The PFID and the HADD are exchanged during the enumeration procedure.

Enumeration is basically the process where the host adds the peripheral to the list of items that it 'knows'.



AGC (2 BIT TIMES)	PRE (5 BIT TIMES)	STA = STS (5 BIT TIMES)	MAC FRAME	CRC = CRC - 8 (8 BITS)	STO (4 BIT TIMES)

(a) SHORT PACKET

AGC (2 BIT TIMES)	PRE (5 BIT TIMES)	STA = STL (5 BIT TIMES)	MAC FRAME	CRC = CRC - 16 (16 BITS)	STO (4 BIT TIMES)

(b) LONG PACKET

Figure 5. System Packet Structure

An IrDA Control peripheral must be enumerated (and bound) with a host before it can exchange data with the host side application layer. A peripheral that has not been enumerated must not perform any communication other than the enumeration procedure. The host ignores a hailing response received from any peripheral to which it has not enumerated.

Special mechanisms may be required on IrDA Control devices to initiate the enumeration procedure, such as a button located on the device. The enumeration procedure uses short frames only and is carried out in the following steps. An example of this is if a new device asks for service when the host is not hailing for new devices, but servicing devices already enumerated and bound.

Peripheral address '0xF' is used in the process of enumeration. During enumeration, the host polls using a peripheral address '0xF'. Unenumerated peripherals are allowed to respond to host polls with PADD of '0xF' only.

The host issues an enumeration hail with the 'hailing' bit set to '1', and a peripheral address 0xF. This host poll frame includes information about the host (Host ID and Host Info).

After storing the HADD, HostID and Host Info data, a peripheral that desires enumeration responds to the hail frame with a frame including its PFID and information about itself, Peripheral Info.

The Peripheral Information tells the host whether the peripheral is a critical latency peripheral (i.e., the peripheral supports the CL polling rate) or not, as well as whether the peripheral has the ability to send or receive long frames. Other information that can be sent as part of the Peripheral Information is a Device Descriptor, Configuration Descriptor and other fields that are used to tell the host more about the peripheral and what it is expected to do. This information can be used to tell the host which device driver is to be used.

The host, which has received the response frame, stores the PFID and Peripheral Information. Then in the next polling cycle, it responds to the peripheral with a frame including the received PFID.

Once the enumeration process is completed, the PFID will be added to the enumeration list in the host. Any item that has been enumerated will be in the list, up to a total of eight items. When additional items are enumerated, the least active device will be dropped from the list. As peripherals are brought into use, the list will be updated for those devices that are in use and have been recently used.

The enumeration procedure may fail due to multiple peripherals responding to the same hail. After responding to the enumeration hail, the peripheral should receive a response from the host with PFID. If a peripheral does not receive the above packet from the host within 69 ms after a request, the peripheral recognizes the failure and goes back to responding with a frame that includes its PFID and a random back-off value between 0 to 7.

If the random back-off value is 0, this peripheral will send a response frame in the next hailing cycle. If the random back-off value is 7, this peripheral will ignore 7 hailing frames and can send a response frame in the eighth hailing cycle.

The full detail of all possible modes and various conditions are in the IrDA Control specification. *

NOTE: *Refer to the complete IrDA Control 1.0 specification, Copyright Infrared Data Association.

The process in which a host dynamically recognizes that an enumerated peripheral needs to be added to the active device-polling loop is called 'Binding'. When bound, the host will include the peripheral device in the active polling cycle and issue poll requests to the device on a cyclic basis. To bind, a process similar to the enumeration sequence is used.

When bound, the peripheral will respond to host polls indicating that it has data for the host. The host will then ask for the data.

When a bound peripheral does not respond to polling for a certain time period, the host recognizes that the peripheral does not need further communication and drops it from the active polling list. This process is called 'Unbinding.' An unbound peripheral is still enumerated and can be picked up into the polling cycle at any time.

When the device has been unbound and sits idle, the peripheral will go into a sleep state where power consumption is very low, typically 1 µA. If we use the mouse as an example, it goes to sleep once it sits idle for more than a few seconds. When asleep, it will awaken when moved or one of the buttons is pushed. Then it will respond to a hailing poll, or will send a wake frame to the host if it is asleep.

The complete details of all cases for binding and unbinding are in the complete specification.*

NOTE: *Refer to the complete IrDA Control 1.0 specification, Copyright Infrared Data Association.

PROTOCOL STACK

The IrDA Protocol stack resides on top of the MAC layer and services the Human Input Device (HID) definition device LLC, the HA LLC and the future device LLCs (yet to be defined). This stack is shown in Figure 6.

At the present time, the HID stack is the most complete and is used in conjunction with standard Universal Serial Bus (USB) definition HID devices.

The stack is basically the same for both the host and peripheral side. In the case of the peripheral, only one of the three application stack LLC columns would be used to service the inputs from the device. In the case of a special function device, more than one column may be present, however the complexity of the peripheral device increases drastically.

In the case of the host, the HID drivers are compatible with USB logical devices. The USB Host version interfaces will implement a Common Class driver that links in with Windows 98* USB drivers to provide an interface to the PC system drivers. The Common Class driver definition allows for multiple devices to work through the single device endpoint in the system, without requiring multiple Serial Interface Engines in the host interface device.

NOTE: *Windows 98 is a trademark of Microsoft Corporation.

When the USB host recognizes a bound peripheral, the device driver information will be passed up to the operating system to identify the bound device. The host machine, when operating with Windows 98, will then ask for the device description and the host will pass this request out to the peripheral.

The peripheral will respond with its peripheral information, which will be passed up to the operating system. The operating system will then load the appropriate driver to service the peripheral.

When a host device is embedded in a system, which has its own operating software, the same process is followed. In this case there may be a variety of drivers to support multiple types of peripheral devices. The operating system will need to provide the same service capabilities, however the USB interface may not be present. If the USB interface is present, then the system software should support the standard USB services.

An example of such an implementation is a Set Top Box (STB) used in the family room. The addition of a keyboard and mouse for use with an Electronic Program Guide (EPG) may be a desired feature set. The STB may have a packaged operating system or use its own software. In either case, all of the AMC and LLC functions must be supported to provide service to the peripheral devices. The STB acts as the host and polls the room.

In a STB application, a forced enumeration may be a desired function. An example is the use a specific keyboard and mouse combination product. The user would not want to enumerate every device that came into view, such as another keyboard or a series of game controllers and joysticks. The user may prefer to tell the STB when a new device should be enumerated instead of hailing the world at large. In either case, the same enumeration and bind process as described above would apply.

The operating system of a STB is likely to talk directly with the IrDA Control host controller and not rely on a USB interface due to the additional cost. It also does not make fiscal sense to add another interface only to speak to an embedded IC. However, depending on feature set, the STB may include a USB port on the rear, so that an IrDA Control interface can be added at a later time, depending on user functions and feature set. In this case, a standard USB host interface could be plugged into the port and function as part of the overall system. The STB operating system would need to support USB services. In STBs that incorporate Windows CE*, some USB support may be provided as part of the operating system in the future, and adding an IrDA Control feature would not be difficult.

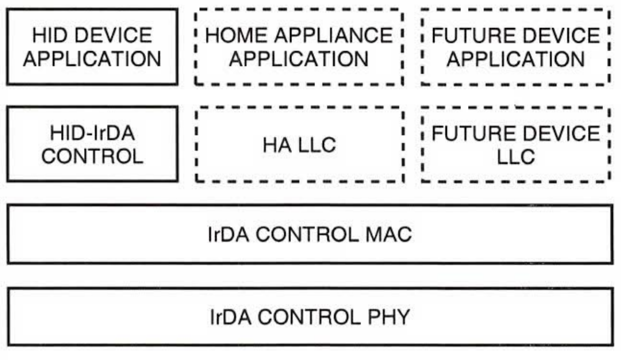
NOTE: *Windows CE is a trademark of Microsoft Corporation.

The Home Appliance application column is still being defined. The Future device application column provides for future devices not yet developed or defined.

LOGICAL LINK CONTROL (LLC)

The Logical Link Control Layer provides resources for reliable communication of data between the MAC layer and the application, as shown in Figure 6. IrDA Control has a goal of providing components, and a protocol, that allow it to be used in a wide range of devices, with great variation in resource and cost requirements.

The LLC provides the link layer resources used by IrDA Control devices, regardless of what higher level protocol may be used. It enables reliability through the use of lightweight protocol controlling frames.



NOTE: Items in dashed boxes indicate future devices.

Figure 6. Protocol Stack

The LLC layer specifies only simple methods for acknowledgment of delivery. Therefore, it might happen that the LLC Layer by itself couldn't honor an application that requires strictly reliable data communication. The upper layers should implement error correction functions, re-transmission functions, and so on, when assurance of reliable communication is necessary. In some cases, such as USB-HID, much of the Link Layer actually resides with the Host Operating System, and the IrDA Control LLC layer is used as a bridge to and from the MAC layer.

The LLC Layer is not utilized during enumeration and binding procedures. The functions of LLC Layer are:

Information Features

- Send Commands
- Receive Requests
- Send Data
- Receive Data

Reliability Features

- Prevent duplicate frames.
- Acknowledgment of delivery based on single frame transmission (ACK).
- Re-transmission function responding to NAK or ignore.
- Provide notice of unsupported features or inability to handle a request at this time.

The LLC field in the MAC frame has control and payload sections that are used to communicate information between layers. Mode and status fields pass bits indicating the packet type and operation to be conducted.

END-USER PRODUCTS

The end goal of the IrDA Control system is to allow developers to create host and peripheral products that are basically interchangeable. A mouse designed for one system should work on all systems, regardless of the manufacturer.

In like manner, any host that supports a mouse, regardless if it is on a PC operating system or an embedded controller product running in a consumer product should recognize an IrDA Control mouse.

Part of the descriptor fields in the peripheral product is an explanation of what it is, so that the correct driver is loaded. Game controllers is another category could experience quick growth, as there is already a large number and variety of products on the market. Drivers for most of these are common on systems or are easily loaded. A correctly implemented IrDA Control device appears transparent to the overall operating system.

One device that is envisioned for the Home Automation LLC path is intelligent remote controls for consumer products. Instead of six remotes on the coffee table, a single remote that could talk to every device in your home. It could be able to download the EPG from the STB and program the VCR and other products. If it has a touch-sensitive Liquid Crystal Display, it could be used by anyone. The size and format of the display could then be customized to meet user needs.

SUMMARY

IrDA Control is a polled host system that supports a variety of cordless input peripherals. It is a control input oriented system that eliminates the wires required for standard input devices. It is appropriate for applications where the user prefers to eliminate the tether and step back from the product in use.

IrDA Control input products are also appropriate in applications where the user prefers to eliminate the clutter of cables in a short-range application.

This paper outlined the hardware approach for the IrDA Control system as well as the Media Access Control and Logical Link Control layers. The MAC controls the software coordination between the hardware and software for system operation. The LLC layer arbitrates the link and tries to maintain an orderly flow of information.

The IrDA Control system can be implemented into a wide variety of products. It is currently PC-oriented, however it does not have to be. Anything that you can envision is a possible application, when you are ready to remove the cable for control input devices.

LIFE SUPPORT POLICY

SHARP components should not be used in medical devices with life support functions or in safety equipment (or similiar applications where component failure would result in loss of life or physical harm) without the written approval of an officer of the SHARP Corporation.

LIMITED WARRANTY

SHARP warrants to its Customer that the Products will be free from defects in material and workmanship under normal use and service for a period of one year from the date of invoice. Customer's exclusive remedy for breach of this warranty is that SHARP will either (i) repair or replace, at its option, any Product which fails during the warranty period because of such defect (if Customer promptly reported the failure to SHARP in writing) or, (ii) if SHARP is unable to repair or replace, refund the purchase price of the Product upon its return to SHARP. This warranty does not apply to any Product which has been subjected to misuse, abnormal service or handling, or which has been altered or modified in design or construction, or which has been serviced or repaired by anyone other than Sharp. The warranties set forth herein are in lieu of, and exclusive of, all other warranties, express or implied. ALL EXPRESS AND IMPLIED WARRANTIES, INCLUDING THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR USE AND FITNESS FOR A PARTICULAR PURPOSE, ARE SPECIFICALLY EXCLUDED. In no event will Sharp be liable, or in any way responsible, for any incidental or consequential economic or property damage.

The above warranty is also extended to Customers of Sharp authorized distributors with the following exception: reports of failures of Products during the warranty period and return of Products that were purchased from an authorized distributor must be made through the distributor. In case Sharp is unable to repair or replace such Products, refunds will be issued to the distributor in the amount of distributor cost.

SHARP reserves the right to make changes in specifications at any time and without notice. SHARP does not assume any responsibility for the use of any circuitry described; no circuit patent licenses are implied.

SHARP®

NORTH AMERICA

SHARP Microelectronics
of the Americas
5700 NW Pacific Rim Blvd., M/S 20
Camas, WA 98607, U.S.A.
Phone: (360) 834-2500
Telex: 49608472 (SHARPCAM)
Facsimile: (360) 834-8903
http://www.sharpsma.com

EUROPE

SHARP Electronics (Europe) GmbH
Microelectronics Division
Sonninstraße 3
20097 Hamburg, Germany
Phone: (49) 40 2376-2286
Facsimile: (49) 40 2376-2232
http://www.sharpmed.com

ASIA

SHARP Corporation
Integrated Circuits Group
2613-1 Ichinomoto-Cho
Tenri-City, Nara, 632, Japan
Phone: (07436) 5-1321
Telex: LABOMETA-B J63428
Facsimile: (07436) 5-1532

>> Forum: **w.palm.dev.forum-l**
>> Thread: **A chat program using Irda between Palm and PC**
>> Message 3 of 5

MY Save this thread

back to search results

Subject: **RE: A chat program using Irda between Palm and PC**
Date: 01/07/2000
Author: **Schettino, John** <schettj@exch.hpl.hp.com>

<< previous in search · next in search >>

Irda is not fun on the Palm...

- swap the two lines

```
err = IrBind(refNum, &IrCon, IrHandler);
IrSetDeviceInfo(refNum, MyDeviceInfo, MyDeviceInfoLen);
```

to

```
IrSetDeviceInfo(refNum, MyDeviceInfo, MyDeviceInfoLen);
err = IrBind(refNum, &IrCon, IrHandler);
```

(set device info before you bind)

Now for the bad news... You're not going to be able to do the discovery right after the bind, because the media will be busy for a little while. You can try waiting for it to go non-busy:

```
while (IrIsMediaBusy (refNum))
SysTaskDelay(SysTicksPerSecond()/10));
```

That gives the Irda stack a chance to come up. Then you can do the irStatus = IrDiscoverReq(refNum, &IrCon);

==-
John Schettino author of
Palm OS Programming For Dummies, http://schettino.tripod.com

-----Original Message-----
From: Matchz [mailto:matchz@i.am]
Sent: Thursday, January 06, 2000 7:41 PM
To: palm-dev-forum@3com.com
Subject: A chat program using Irda between Palm and PC

I am trying to write a chat program using Irda between a Palm and PC. On the PC side, I use winsock2. I initialize the sock as follow:

```
SOCKET cli_sock = socket(AF_IRDA, SOCK_STREAM, 0);
SOCKADDR_IRDA srv_addr = {AF_IRDA, 0, 0, 0, 0, "PALMDEMO"};
bind(cli_sock,(LPSOCKADDR)&cli_addr,sizeof(cli_addr));
```

The code work well for 2 PC both have an Irda port.

So I try to program the palm side. I start the Irda service as follow:

```
static UInt refNum;
static IrConnect IrCon;

static Byte MyDeviceInfo[] = {IR_HINT_PDA, IR_CHAR_ASCII,
  ’P’,’A’,’L’,’M’,’D’,’E’,’M’,’O’};
static Byte MyDeviceInfoLen = sizeof(MyDeviceInfo);

err = SysLibFind(irLibName, &refNum);
err = IrOpen(refNum, irOpenOptSpeed9600);
err = IrBind(refNum, &IrCon, IrHandler);
IrSetDeviceInfo(refNum, MyDeviceInfo, MyDeviceInfoLen);

irStatus = IrDiscoverReq(refNum, &IrCon);
switch (irStatus) {
case IR_STATUS_MEDIA_BUSY:
 SetStatus("IR Status Media Busy");
 break;
case IR_STATUS_FAILED:
 SetStatus("IR Status Failed");
 IrUnbind(refNum, &IrCon);
 IrClose(refNum);
 break;
case IR_STATUS_PENDING:
 SetStatus("IR Status Pending => Success");
 handled = true;
}
```

However, function IrDisvocerReq always return IR_STATUS_MEDIA_BUSY while I expect a IR_STATUS_PENDING should be returned. We can assume that the statements before IrDisvoverReq always return no error.

Do I make any mistake in the statements above, what should I pay attention
when I
programming Irda?

Thank you.
Matchz.

<< previous in search · next in search >>

Subscribe to **w.palm.dev.forum-l**
Mail this message to a friend
View original Usenet format
Create a custom link to this message from your own Web site

Search Discussions For a more detailed search in Discussions go to Power Search

Search only in: w.palm.dev.forum-l
All Deja.com

Search for: `IrConnect`

Search recent
Search discussions

Advertise With Us | About Deja.
Trademarks · Terms and Conditions of Use · Site Privacy Statement

>> Forum: **w.palm.dev.forum-l**
>> Thread: **A chat program using Irda between Palm and PC**
>> Message 4 of 5

Save this thread

back to search results

Subject: **Re: A chat program using Irda between Palm and PC**
Date: 01/07/2000
Author: **Tom Frauenhofer** <frauen1@yahoo.com>

<< previous in search · next in search >>

You get a lot of MEDIA_BUSY's from the IR - put the IrDiscoverReq()/switch() line in a loop, have it end the loop if it doesn't get a MEDIA_BUSY (either the PENDING or the ERROR should stop the loop).

(BTW, a good resource for IR programming at this level is Glen Bachmann's Palm Programming book - I believe he even has the chapter on IR programming on his web site at http://www.bachmannsoftware.com/)

At 11:41 AM 1/7/00 +0800, you wrote:

```
>I am trying to write a chat program using Irda between a Palm and PC.
>On the PC side, I use winsock2. I initialize the sock as follow:
>
>SOCKET cli_sock = socket(AF_IRDA, SOCK_STREAM, 0);
>SOCKADDR_IRDA srv_addr = {AF_IRDA, 0, 0, 0, 0, "PALMDEMO"};
>bind(cli_sock,(LPSOCKADDR)&cli_addr,sizeof(cli_addr));
>
>The code work well for 2 PC both have an Irda port.
>
>So I try to program the palm side. I start the Irda service as follow:
>
> static UInt refNum;
> static IrConnect IrCon;
>
> static Byte MyDeviceInfo[] = {IR_HINT_PDA, IR_CHAR_ASCII,
>      'P','A','L','M','D','E','M','O'};
> static Byte MyDeviceInfoLen = sizeof(MyDeviceInfo);
```

```
>
>err = SysLibFind(irLibName, &refNum);
>err = IrOpen(refNum, irOpenOptSpeed9600);
>err = IrBind(refNum, &IrCon, IrHandler);
>IrSetDeviceInfo(refNum, MyDeviceInfo, MyDeviceInfoLen);
>
>irStatus = IrDiscoverReq(refNum, &IrCon);
>switch (irStatus) {
>       case IR_STATUS_MEDIA_BUSY:
>               SetStatus("IR Status Media Busy");
>               break;
>       case IR_STATUS_FAILED:
>               SetStatus("IR Status Failed");
>               IrUnbind(refNum, &IrCon);
>               IrClose(refNum);
>               break;
>       case IR_STATUS_PENDING:
>               SetStatus("IR Status Pending => Success");
>               handled = true;
>}
>
>However, function IrDisvocerReq always return
IR_STATUS_MEDIA_BUSY while I
>expect a IR_STATUS_PENDING should be returned. We can assume that the
>statements before IrDisvoverReq always return no error.
>
>Do I make any mistake in the statements above, what should I pay attention
>when I
>programming Irda?
>
>Thank you.
>Matchz.
```

<< previous in search · **next in search >>**

Subscribe to **w.palm.dev.forum-l**
Mail this message to a friend
View original Usenet format
Create a custom link to this message from your own Web site

Search Discussions For a more detailed search in Discussions go to Power Search

Search only in: w.palm.dev.forum-l
All Deja.com

Search for: `IrConnect`

Search

Search recent discussions

>> Forum: **w.palm.dev.forum-l**
>> Thread: **Proper format for XID string in IrSetDeviceInfo?**
>> Message 1 of 5

MY Save this thread

back to search results

Subject: **Re: Proper format for XID string in IrSetDeviceInfo?**
Date: 12/18/1999
Author: **Danny Epstein** <Danny@AppliedThought.com>

<< previous · next in search >>

At 9:57 AM -0800 12/16/99, Schettino, John wrote:
>I'm attempting to use the IrSetDeviceInfo() call to set the device info for
>an IRDA app... What's the correct format for the XID string?
>
>This isn't it: IrSetDeviceInfo (ir_ref, (BytePtr) "Palm eSquirt", 12);

By convention, the first one or two bytes are hints with all but the last hint byte having the high bit (IR_HINT_EXT) set. The remainder of the XID is up to you.

The exchange manager uses IR_HINT_PDA | IR_HINT_EXT for the first byte, IR_HINT_OBEX for the second, and the HotSync username for the remainder. In BeamBooks, we don't set IR_HINT_OBEX and we include the creator ID of our app rather than the HotSync username so that the other side can confirm it's talking to a peer (see below). We also include the local LSAP selector, which is something you'll probably want to do.

When you get a discover confirmation, you can go through the XIDs of the discovered devices to pick who you want to connect with, if anyone. We look for an XID just like the one we'd generate, ignoring the LSAP selector. This ensures we're talking to another device running our app (possibly on some other kind of hardware). If we find a device that has the same XID as we do (ignoring the LSAP selector), we grab the LSAP selector and store it in the rLsap field of the **IrConnect** structure.

AFAIK there is no central registry for XIDs.
-

Danny Epstein, Applied Thought Corporation
Have friends with Palm organizers? Get BeamBooks!
http://www.appliedthought.com/beambooks

<< previous · next in search >>

Subscribe to **w.palm.dev.forum-l**
Mail this message to a friend
View original Usenet format
Create a custom link to this message from your own Web site

Search Discussions For a more detailed search in Discussions go to Power Search

Search only in: w.palm.dev.forum-l
All Deja.com

Search for: `IrConnect`

Search recent · Search discussions

Advertise With Us | About Deja.
Trademarks · Terms and Conditions of Use · Site Privacy Statement

```
// name of Ir library
#define irLibName "IrDA Library"

// Feature Creators and numbers, for use with the FtrGet() call. This
//  feature can be obtained to get the current version of the Ir Library
#define irFtrCreator            sysFileCIrLib
#define irFtrNumVersion         0             // get version of Net Library
        // 0xMMmfsbbb, where MM is major version, m is minor version
        // f is bug fix, s is stage: 3-release,2-beta,1-alpha,0-development,
        // bbb is build number for non-releases
        // V1.12b3   would be: 0x01122003
        // V2.00a2   would be: 0x02001002
        // V1.01     would be: 0x01013000


// Options values for IrOpen
#define irOpenOptSpeed115200    0x0000003F    // sets max negotiated baud rate
#define irOpenOptSpeed57600     0x0000001F    // default is 57600
#define irOpenOptSpeed38400     0x0000000F
#define irOpenOptSpeed19200     0x00000007
#define irOpenOptSpeed9600      0x00000003

// Option codes for ExgLibControl
// These options are all unique to the Ir transport:

// returns scanning enabled
#define irGetScanningMode (exgLibCtlSpecificOp | 1)

// en/disables ir scanning mode
#define irSetScanningMode (exgLibCtlSpecificOp | 2)

// returns performance stats
#define irGetStatistics   (exgLibCtlSpecificOp | 3)

// sets driver to use direct serial
#define irSetSerialMode   (exgLibCtlSpecificOp | 4)

// set possible baud rates (irOpenOpts)
#define irSetBaudMask     (exgLibCtlSpecificOp | 5)

// disables the ir not supported dialog
#define irSetSupported    (exgLibCtlSpecificOp | 6)


// structure returned by irGetStatistics
typedef struct {
    Word recLineErrors;        // # of serial errors since library opend
    Word crcErrors;            // # of crc errors ...
} IrStatsType;

//-----------------------------------------------------------------------
// Ir library call ID's.
//-----------------------------------------------------------------------
#pragma mark Traps
typedef enum {
    irLibTrapBind = exgLibTrapLast,  // these start after the ObxLib interface...
    irLibTrapUnBind,
    irLibTrapDiscoverReq,
    irLibTrapConnectIrLap,
    irLibTrapDisconnectIrLap,
    irLibTrapConnectReq,
    irLibTrapConnectRsp,
    irLibTrapDataReq,
    irLibTrapLocalBusy,
    irLibTrapMaxTxSize,
    irLibTrapMaxRxSize,
    irLibTrapSetDeviceInfo,
    irLibTrapIsNoProgress,
    irLibTrapIsRemoteBusy,
    irLibTrapIsMediaBusy,
    irLibTrapIsIrLapConnected,
    irLibTrapTestReq,
    irLibTrapIAS_Add,
    irLibTrapIAS_Query,
    irLibTrapIAS_SetDeviceName,
    irLibTrapIAS_Next,
    irLibTrapIrOpen,
    irLibTrapHandleEvent,
    irLibTrapWaitForEvent,

    irLibTrapLast
} IrLibTrapNumberEnum;

/****************************************************************************
 *
 * Types and Constants
 *
 ****************************************************************************/

/* Maximum size of packet that can be sent at connect time (ConnectReq or
 * ConnectRsp) for IrLMP and Tiny TP connections.
 */
#define IR_MAX_CON_PACKET     60
#define IR_MAX_TTP_CON_PACKET 52
#define IR_MAX_TEST_PACKET    376
#define IR_MAX_DEVICE_INFO    23

/* Size of the device list used in discovery process
 */
#define IR_DEVICE_LIST_SIZE 6

/*---------------------------------------------------------------------------
 *
 * Maximum size of the XID info field used in a discovery frame. The XID
 * info field contains the device hints and nickname.
 */
#define IR_MAX_XID_LEN   23


/* Maximum allowed LSAP in IrLMP
 */
#define IR_MAX_LSAP      0x6f

/* The following are used to access the hint bits in the first byte
 * of the Device Info field of an XID frame (IrDeviceInfo).
 */
#define IR_HINT_PNP       0x01
#define IR_HINT_PDA       0x02
#define IR_HINT_COMPUTER  0x04
#define IR_HINT_PRINTER   0x08
#define IR_HINT_MODEM     0x10
#define IR_HINT_FAX       0x20
#define IR_HINT_LAN       0x40
#define IR_HINT_EXT       0x80

/* The following are used to access the hint bits in the second byte
 * of the Device Info field of an XID frame (IrDeviceInfo). Note
 * that LM_HINT_EXT works for all hint bytes.
 */
#define IR_HINT_TELEPHONY 0x01
#define IR_HINT_FILE      0x02
#define IR_HINT_IRCOMM    0x04
#define IR_HINT_MESSAGE   0x08
#define IR_HINT_HTTP      0x10
```

```
#define IR_HINT_OBEX      0x20


/*---------------------------------------------------------------------------
 *
 * Status of a stack operation or of the stack.
 */
typedef Byte IrStatus;

#define IR_STATUS_SUCCESS         0  /* Successful and complete */
#define IR_STATUS_FAILED          1  /* Operation failed */
#define IR_STATUS_PENDING         2  /* Successfully started but pending */
#define IR_STATUS_DISCONNECT      3  /* Link disconnected */
#define IR_STATUS_NO_IRLAP        4  /* No IrLAP Connection exists */
#define IR_STATUS_MEDIA_BUSY      5  /* IR Media is busy */
#define IR_STATUS_MEDIA_NOT_BUSY  6  /* IR Media is not busy */
#define IR_STATUS_NO_PROGRESS     7  /* IrLAP not making progress */
#define IR_STATUS_LINK_OK         8  /* No progress condition cleared */

/*---------------------------------------------------------------------------
 *
 * Character set for user strings. These are definitions for the character
 * set in Nicknames and in IAS attributes of type User String.
 */
typedef Byte IrCharSet;

#define IR_CHAR_ASCII       0
#define IR_CHAR_ISO_8859_1  1
#define IR_CHAR_ISO_8859_2  2
#define IR_CHAR_ISO_8859_3  3
#define IR_CHAR_ISO_8859_4  4
#define IR_CHAR_ISO_8859_5  5
#define IR_CHAR_ISO_8859_6  6
#define IR_CHAR_ISO_8859_7  7
#define IR_CHAR_ISO_8859_8  8
#define IR_CHAR_ISO_8859_9  9
#define IR_CHAR_UNICODE     0xff

/*---------------------------------------------------------------------------
 *
 * All indication and confirmations are sent to the IrLMP/TTP connections
 * through one callback function. The types of the events passed are
 * defined below.
 */
typedef Byte IrEvent;

#define LEVENT_LM_CON_IND     0
#define LEVENT_LM_DISCON_IND  1
#define LEVENT_DATA_IND       2
#define LEVENT_PACKET_HANDLED 3
#define LEVENT_LAP_CON_IND    4
#define LEVENT_LAP_DISCON_IND 5
#define LEVENT_DISCOVERY_CNF  6
#define LEVENT_LAP_CON_CNF    7
#define LEVENT_LM_CON_CNF     8
#define LEVENT_STATUS_IND     9
#define LEVENT_TEST_IND       10
#define LEVENT_TEST_CNF       11


/* LmConnect flags - used internally
 */
#define LCON_FLAGS_TTP      0x02


/****************************************************************************
 *
 * IAS Types and Constants
 *
 ****************************************************************************/

/* Maximum size of a query that observes the IrDA Lite rules
 */
#define IR_MAX_QUERY_LEN 61

/* Maximum values for IAS fields. IR_MAX_IAS_NAME is the maximum allowable
 * size for IAS Object names and Attribute names.
 */
#define IR_MAX_IAS_NAME           60
#define IR_MAX_ATTRIBUTES        255

/* Maximum size of an IAS attribute that fits within the IrDA Lite rules.
 * Even though attribute values can be larger IrDA Lite highly recommends
 * that the total size of an attribute value fit within one 64 byte packet
 * thus, the allowable size is 56 bytes or less. This size is enforced by the
 * code.
 */
#define IR_MAX_IAS_ATTR_SIZE      56

/* Type of the IAS entry. This is the value returned for type when parsing
 * the results buffer after a successful IAS Query.
 */
#define IAS_ATTRIB_MISSING      0
#define IAS_ATTRIB_INTEGER      1
#define IAS_ATTRIB_OCTET_STRING 2
#define IAS_ATTRIB_USER_STRING  3
#define IAS_ATTRIB_UNDEFINED    0xff

/* Ias Return Codes. One of these values will be found in the IAS Query
 * structure in the retCode field after a successful IAS Query.
 */
#define IAS_RET_SUCCESS        0    /* Query operation is successful */
#define IAS_RET_NO_SUCH_CLASS  1    /* Query failed no such class exists */
#define IAS_RET_NO_SUCH_ATTRIB 2    /* Query failed no such attribute exists */
#define IAS_RET_UNSUPPORTED    0xff /* Query failed operation is unsupported */

 /* IAS Get Value By Class opcode number
  */
#define IAS_GET_VALUE_BY_CLASS 4

// Macros used in accessing ias structures
#define IasGetU16(ptr) (Word)( ((Word)(*((Byte*)ptr) << 8)) | \
                       ((Word) (*((Byte*)ptr+1))))
#define IasGetU32(ptr) (DWord)( ((DWord)(*((Byte*)ptr)) << 24) | \
                              ((DWord)(*((Byte*)ptr+1)) << 16) | \
                              ((DWord)(*((Byte*)ptr+2)) << 8)  | \
                              ((DWord)(*((Byte*)ptr+3))) )

/****************************************************************************
 *
 * Data Structures
 *
 ****************************************************************************/

// stack functions use a diferent type for booleans
typedef int BOOL;

/*---------------------------------------------------------------------------
 *
 * ListEntry is used internally by the stack
 */
typedef struct  _ListEntry
{
    struct _ListEntry *Flink;
    struct _ListEntry *Blink;

} ListEntry;

/* Forward declaration of the IrConnect structure
 */
typedef struct _hconnect IrConnect;

/*---------------------------------------------------------------------------
 *
 * Packet Structure for sending IrDA packets.
 */
typedef struct _IrPacket {
    /* The node field must be the first field in the structure. It is used
     * internally by the stack
     */
    ListEntry  node;

    /* The buff field is used to point to a buffer of data to send and len
     * field indicates the number of bytes in buff.
     */
    BytePtr   buff;
    Word      len;

    /*==================  For Internal Use Only ======================
     *
     * The following is used internally by the stack and should not be
     * modified by the upper layer.
     *
     *==============================================================*/

    IrConnect* origin;     /* Pointer to connection which owns packet */
    Byte       headerLen;  /* Number of bytes in the header */
    Byte       header[14]; /* Storage for the header */
} IrPacket;

/*---------------------------------------------------------------------------
 *
 * 32-bit Device Address
 */
typedef  union {
    Byte  u8[4];
    Word  u16[2];
    DWord u32;
} IrDeviceAddr;

/*---------------------------------------------------------------------------
 *
 * The information returned for each device discovered during discovery.
 * The maximum size of the xid field is 23. This holds the hints and
 * the nickname.
 */
typedef  struct {
    IrDeviceAddr hDevice;             /* 32-bit address of device */
    Byte         len;                 /* Length of xid */
    Byte         xid[IR_MAX_XID_LEN];/* XID information */
} IrDeviceInfo;

/*---------------------------------------------------------------------------
 *
 * List of Device Discovery info elements.
 */
typedef  struct {
  Byte          nItems;                     /* Number items in the list */
  IrDeviceInfo  dev[IR_DEVICE_LIST_SIZE]; /* Fixed size in IrDA Lite */
```

```
} IrDeviceList;

/*---------------------------------------------------------------------------
 *
 * Callback Parameter Structure is used to pass information from the stack
 * to the upper layer of the stack (application). Not all fields are valid
 * at any given time. The type of event determines which fields are valid.
 */
typedef struct {
    IrEvent       event;       /* Event causing callback */
    BytePtr       rxBuff;      /* Receive buffer already advanced to app data */
    Word          rxLen;       /* Length of data in receive buffer */
    IrPacket*     packet;      /* Pointer to packet being returned */
    IrDeviceList* deviceList;  /* Pointer to discovery device list */
    IrStatus      status;      /* Status of stack */
} IrCallBackParms;

/* The definitions for the callback function is given below. How the
 * callback function is used in conjuction with the stack functions is
 * given below in the Callback Reference.
 */
typedef void (*IrCallBack)(IrConnect*, IrCallBackParms*);

/*---------------------------------------------------------------------------
 *
 * Definition of IrConnect structure. This structure is used to manage an
 * IrLMP or Tiny TP connection.
 */
typedef struct _hconnect {
    Byte         lLsap;      /* Local LSAP this connection will listen on */
    Byte         rLsap;      /* Remote Lsap */

    /*==================  For Internal Use Only ======================
     *
     * The following is used internally by the stack and should not be
     * modified by the user.
     *
     *==============================================================*/

    Byte         flags;      /* Flags containing state, type, etc. */
    IrCallBack   callBack;   /* Pointer to callback function */

    /* Tiny TP fields */
    IrPacket     packet;      /* Packet for internal use */
    ListEntry    packets;     /* List of packets to send */
    Word         sendCredit;  /* Amount of credit from peer */
    Byte         availCredit; /* Amount of credit to give to peer */
    Byte         dataOff;     /* Amount of data less than IrLAP size */
} _hconnect; IrConnect

/****************************************************************************
 *
 * IAS Data Strucutres
 *
 ****************************************************************************/

/*---------------------------------------------------------------------------
 *
 * The LmIasAttribute is a strucutre that holds one attribute of an IAS
 * object.
 */
typedef struct _IrIasAttribute {
    BytePtr       name;    /* Pointer to name of attribute */
    Byte          len;     /* Length of attribute name */
    BytePtr       value;   /* Hardcode value (see below) */
    Byte          valLen;  /* Length of the value. */
} IrIasAttribute;
```

```
/* The value field of the IrIasAttribute structure is a hard coded string
 * which represents the actual bytes sent over the IR for the attribute
 * value. The value field contains all the bytes which represent an
 * attribute value based on the transmission format described in section
 * 4.3 of the IrLMP specification. An example of a user string is given
 * below.
 *
 * User String:
 *   1 byte type,  1 byte Char set, 1 byte length, length byte string
 *
 * Example of an user string "Hello World" in ASCII
 *
 * U8 helloString[] = {
 *    IAS_ATTRIB_USER_STRING,IR_CHAR_ASCII,11,
 *    'H','e','l','l','o',' ','W','o','r','l','d'
 * };
 */

/*---------------------------------------------------------------------------
 *
 * The LmIasObject is storage for an IAS object managed by the local
 * IAS server.
 */
typedef struct _IrIasObject {
  BytePtr         name;      /* Pointer to name of object */
  Byte            len;       /* Length of object name */

  Byte            nAttribs;  /* Number of attributes */
  IrIasAttribute* attribs;   /* A pointer to an array of attributes */

} IrIasObject;


/*---------------------------------------------------------------------------
 *
 * Forward declaration of a structure used for performing IAS Queries so
 * that a callback type can be defined for use in the structure.
 */
typedef struct _IrIasQuery IrIasQuery;
typedef void (*IrIasQueryCallBack)(IrStatus);

/*---------------------------------------------------------------------------
 *
 * Actual definition of the IrIasQuery structure.
 */
typedef struct _IrIasQuery
{
    /* Query fields. The query buffer contains the class name and class
     * attribute whose value is being queried it is as follows:
     *
     * 1 byte - Length of class name
     * "Length" bytes - class name
     * 1 byte - length of attribute name
     * "Length" bytes - attribute name
     *
     * queryLen - contains the total number of byte in the query
     */
    Byte     queryLen;       /* Total length of the query */
    BytePtr  queryBuf;       /* Points to buffer containing the query */

    /* Fields for the query result */
    Word     resultBufSize;  /* Size of the result buffer */
    Word     resultLen;      /* Actual number of bytes in the result buffer */
    Word     listLen;        /* Number of items in the result list. */
    Word     offset;         /* Offset into results buffer */
    Byte     retCode;        /* Return code of operation */
    Byte     overFlow;       /* Set TRUE if result exceeded result buffer size */
    BytePtr  result;         /* Pointer to buffer containing result; */

    /* Pointer to callback function */
    IrIasQueryCallBack callBack;
} _IrIasQuery;
```

mIrIasQuery

Beyond-Security's SecuriTeam.com

SecuriTeam Home
About SecuriTeam
Ask the Team
Advertising info
Security News
Security Reviews
Exploits
Tools
UNIX focus
Windows NT focus

Search

Title **28/9/2000**

PalmOS Password Retrieval and Decoding

Summary

PalmOS offers a built-in Security application, which is used for the legitimate user to protect and hide records from unauthorized users by means of a password. In all basic built-in applications (Address, Date Book, Memo Pad, and To Do List), individual records can be marked as "Private" and will only be accessible if the correct password is entered.
It is possible to obtain an encoded form of the password, determine the actual password due to a weak, reversible encoding scheme, and access a users private data. In order for this attack to be successful, the attacker must have physical access to the target Palm device.
The threat of physical attacks internal to a company is very real and this advisory makes the point that security is not limited to the network/internet arena. The private records often contain passwords, financial data, and company confidential information. @stake's experience with physical audits has revealed that most users of Palm or other portable devices do not realize that their private information could possibly be accessed by unauthorized users.

Details

During the HotSync process, the Palm device sends an encoded form of the password over the serial, IR, or network ports to the HotSync Manager or HotSync Network Server on the desktop. The password is transmitted to enable the Palm Desktop program to protect the users private records when being accessed on the desktop machine. However, based on an encoding scheme of XOR'ing against a constant block of data, the encoded password is easily decoded into the actual ASCII version of the password. The encoded block is also stored on the Palm device in the Unsaved Preferences database, readable by any application on the Palm device.

The transfer of a secret component (i.e. password), even if it is encoded or obfuscated, over accessible buses (serial, IR, or network) is a very risky design decision and is oftentimes considered a design flaw. It is an unfortunate common practice for applications to simply obfuscate passwords instead of using encryption. Without proper encryption methodologies in place, the task of determining the secret data is greatly simplified as shown in this research.

This advisory is an attempt to remind users and developers of the common problem of storing secrets and the reliance on simple obfuscation.

1. PHP3/PHP4 Format String vulnerability exposes web servers to machine compromise
2. Siemens HiNet Phone vulnerable to a DoS
3. Cisco PIX Firewall 'SMTP content' fix found to be flawed
4. Scp file transfer hole
5. Cisco Secure PIX Firewall Mailguard Vulnerability (Patch available)

E-Mail this article to a friend
Send us comments

Technical Description:

The password is set by the legitimate user with the Security application. The ASCII password has a maximum length of 31 characters. Regardless of the length of the ASCII password, the resultant encoded block is always 32 bytes.

It is possible to obtain the encoded password block in a number of ways:

1. Retrieve from the "Unsaved Preferences" database on the Palm device.
2. Monitor the serial or network traffic during an actual HotSync.
3. Imitate the initial HotSync negotiation sequence in order to obtain the password (which is transmitted by the target device). This is demonstrated in the proof-of-concept tool written by @stake for the PalmOS platform.

The Palm desktop software makes use of the Serial Link Protocol (SLP) to transfer information between itself and the Palm device. Each SLP packet consists of a packet header, client data of variable size, and a packet footer [Palm OS Programmer's Companion, pg. 255]. During the HotSync negotiation process, one particular SLP packet's client data consists of a structure that contains the encoded password block:

```
struct {
  UInt8 header[4];
  UInt8 exec_buf[6];
  Int32 userID; // 0
  Int32 viewerID; // 4
  Int32 lastSyncPC; // 8
  UInt8 successfulSyncDate[8]; // 12, time_t
  UInt8 lastSyncDate[8]; // 20, time_t
  UInt8 userLen; // 28
  UInt8 passwordLen; // 29
  UInt8 username[128]; // 30 -> userLen
  UInt8 password[128];
};
```

Two methods are used to encode the ASCII password depending on its length. For passwords of 4 characters or less, an index is calculated based on the length of the password and the string is XOR'ed against a 32-byte constant block. For passwords greater than 4 characters, the string is padded to 32 bytes and run through four rounds of a function which XOR's against a 64-byte constant block. It is unknown why disparate methods were implemented. By understanding the encoding schema used, it is possible to essentially run the routines in reverse to decode the password, as shown in our proof-of-concept tools. Details of each method are described below.

Neither encoding schema makes use of the username, user ID, or unique serial number of the Palm device. A common practice often used for copy-protection purposes is to use a unique identifier as input into an encoding or encryption algorithm, which

PalmOS does not do. The resultant encoded password block is completely independent of the Palm device used and makes it easier to determine the original ASCII password from the block.

Passwords of 4 characters or less:

By comparing the encoded password blocks of various short length passwords, it was determined that a 32-byte constant was being XOR'ed against the ASCII password in the following fashion:

```
56 8C D2 3E 99 4B 0F 88 09 02 13 45 07 04 13 44
0C 08 13 5A 32 15 13 5D D2 17 EA D3 B5 DF 55 63
```

Encoded password block of ASCII password 'test'

```
09 02 13 45 07 04 13 44 0C 08 13 5A 32 15 13 5D
D2 17 EA D3 B5 DF 55 63 22 E9 A1 4A 99 4B 0F 88
```

32-byte constant block for use with passwords of length 4 characters or less

Let A_j be the jth byte of A, the ASCII password
Let B_k be the kth byte of B, the 32-byte constant block
Let C_m be the mth byte of C, the encoded password block

The starting index, i, into the constant block where the XOR'ing should begin is calculated by the following:

```
i = (A_0 + strlen(A)) % 32d;
```

The encoded password block is then created:

```
C_0 = A_0 XOR B_i
C_1 = A_1 XOR B_i+1
C_2 = A_2 XOR B_i+2
C_3 = A_3 XOR B_i+3
C_4 = B_i+4
  .
  .
  .
C_31 = B_i+31 (wrapping around to the beginning of the constant
 block if necessary)
```

Example: 0x56 = 0x74 ('t') XOR 0x22
0x8C = 0x65 ('e') XOR 0xE9
0xD2 = 0x73 ('s') XOR 0xA1
0x3E = 0x74 ('t') XOR 0x4A

Passwords greater than 4 characters:

The encoding scheme for long length passwords (up to 31 characters in length) is more complicated than for short length passwords, although it, too, is reversible.

First, the ASCII string is padded to 32 bytes in the following fashion:

Let A_j be the jth byte of A, the ASCII password

```
   len = strlen(A);
   while (len < 32)
   {
     for (i = len; i < len * 2; ++i)
       pass[i] = pass[i - len] + len; // increment each character by
len

     len = len * 2;
   }
```

Example: A_0 = 0x74 ('t')
A_1 = 0x65 ('e')
A_2 = 0x73 ('s')
A_3 = 0x74 ('t')
A_4 = 0x61 ('a')
A_5 = 0x79
A_6 = 0x6A
A_7 = 0x78
A_8 = 0x79
A_9 = 0x66
A_10 = 0x7E
.
.
.

The resultant 32-byte array, A, is then passed through four rounds of a function which XOR's against a 64-byte constant:

```
B1 56 35 1A 9C 98 80 84 37 A7 3D 61 7F 2E E8 76
2A F2 A5 84 07 C7 EC 27 6F 7D 04 CD 52 1E CD 5B
B3 29 76 66 D9 5E 4B CA 63 72 6F D2 FD 25 E6 7B
C5 66 B3 D3 45 9A AF DA 29 86 22 6E B8 03 62 BC
```

Let B_k be the kth byte of B, the 64-byte constant block
Let m = 2, 16, 24, 8 for each of the four rounds

```
index = (A_m + A_m+1) & 0x3F; // 6 LSB
```

```
shift = (A_m+2 + A_m+3) & 0x7; // 3 LSB

for (i = 0; i < 32; ++i)
{
  if (m == 32) m = 0; // wrap around to beginning
  if (index == 64) index = 0; // wrap around to beginning

  temp = B_index; // xy
  temp <<= 8;
  temp |= B_index; // xyxy

  temp >>= shift;
  A_m ^= (unsigned char) temp;

  ++m;
  ++index;
}
```

The resultant 32-byte encoded password block does not have any remnants of the constant block as the short length encoding method does. Although the block appears to be "random", it is indeed reversible with minimal computing resources as shown in our proof-of-concept tools.

```
18 0A 43 3A 17 7D A3 CA D7 9D 75 D2 D3 C8 A5 CF
F1 71 07 03 5A 52 4B B9 70 2D B2 D1 DF A5 54 07
```

Encoded password block of ASCII password ’testa’

Temporary Solution:

The Security application provides functionality to "turn off and lock device". If the Palm device is turned off and locked using this feature, the device will not be operational until the correct password is entered. This will prevent an unauthorized user from running applications on the device (hence preventing them from starting the HotSync process). This workaround is only useful if the legitimate user can be sure that the attacker hasn’t attained the system password already - simply change the password to be sure. It may be possible to bypass the system lock-out mechanism by entering into the PalmOS debug mode before the lock-out features are called. This may allow an attacker to step over the security code during a debugging session.

Another possible solution is the use of third-party encryption solutions, such as Secure Memopad by Certicom, which implement strong and tested cryptological algorithms to protect the data of certain Palm applications.

Vendor Response:

The following is the response @stake received from Palm Inc.

Thanks you for your diligence in testing our products thoroughly, we appreciate your efforts.

We have taken a close look at your advisory in detail and while this is certainly something we want to address for the future, we do not believe this poses a major risk to all our users for the following reasons:
It is not easy for someone to capture passwords accidentally, you need to have access to the device and access to the OS/software as well to run the hotsync and thence capture the data. It would also need to be a malicious, funded, attack and some data points need to be known to the attacker, making the chances of such an attack very low, but not impossible in everyday life. However we do appreciate the risk involved if the attacker is involved in some form of industrial espionage for example.

The simple way to protect against such an attack is to use products from Force.com to keep the device about your person, or to use any of the security programs such as OnlyMe or SignOn to secure access, (as improvements over the supplied software security program) or data encryption programs such as Jaws Technology encryptors, Securememopad from Certicom to encrypt data, or Ntru encryption tools.

However we agree that any potential security issue needs to be taken seriously and we have investigated this problem and expect to have both a patch for older systems, and a solution for future releases of the PalmOS.

We respectfully ask you to post our response with your advisory, and thank you for contributing to the secure future of Palm devices.

Proof-of-Concept Code:

Proof-of-concept tools have been written for the Windows 9x/NT and PalmOS 3.3 and greater platforms which demonstrate the simplicity of obtaining the encoded password block from the target device and the weak encoding scheme used to obfuscate the password. The PC version, "PalmCrypt", will encode and decode ASCII passwords to encoded password blocks and vice versa. The PalmOS version, "NotSync", will imitate the initial stages of the HotSync process via the IR port, retrieve the encoded password block of the target device, and decode and display the
resultant ASCII password.

Source code and binaries for the proof-of-concept tools can be found at:

http://www.atstake.com/research/advisories/2000/notsync.zip
http://www.atstake.com/research/advisories/2000/palmcrypt.zip

Successfully using NotSync requires two Palm devices: One device running the NotSync application and the other being the target device in which the password is desired.

Facing the two devices head-to-head, run the HotSync application on the target Palm device and initiate an "IR to a PC/Handheld" HotSync. NotSync, running on the other device, will obtain the legitimate user's encoded password block, decode the password, and display the result on the screen.

Typical usage and output for PalmCrypt is shown below:

```
E:\>palmcrypt

PalmOS Password Codec
kingpin@atstake.com
@stake Research Labs
http://www.atstake.com/research
August 2000

Usage: palmcrypt -[e | d] <ASCII | password block>

E:\>palmcrypt -e test

PalmOS Password Codec
kingpin@atstake.com
@stake Research Labs
http://www.atstake.com/research
August 2000

0x56 0x8C 0xD2 0x3E 0x99 0x4B 0x0F 0x88  [V..>.K..]
0x09 0x02 0x13 0x45 0x07 0x04 0x13 0x44  [...E...D]
0x0C 0x08 0x13 0x5A 0x32 0x15 0x13 0x5D  [...Z2..]]
0xD2 0x17 0xEA 0xD3 0xB5 0xDF 0x55 0x63  [......Uc]

E:\>palmcrypt -d
568CD23E994B0F8809021345070413440C08135A3215135DD217EAD3B5DF
5563

PalmOS Password Codec
kingpin@atstake.com
@stake Research Labs
http://www.atstake.com/research
August 2000

0x74 0x65 0x73 0x74                 [test    ]
```

Additional information

The information has been provided by @Stake

MCAFEE
THE VACCINE FOR E-BUSINESS

DOWNLOAD UPDATES NOW

PRODUCTS | BUY/TRY | SOLUTIONS | NAI LABS | SERVICES | PARTNERS | ABOUT NAI | NAI HOME

Virus Alerts
Anti-Virus Updates
Virus Library
► Virus Info Center
Recent Updates
Joke Programs
Trojans
Hoaxes
Web Viruses
White Papers

AVERT Research Center
AVERT WebImmune
SecureCast

SITE SEARCH

Profile

Virus Name
PalmOS/Phage.963

Aliases
Palm Virus
Phage 1.0

Variants

Name	Type	Sub Type	Differences
PalmOS/Phage.1325.dr	Virus	PDA Device	This is the initial virus dropper, 1325 bytes, named PHAGE.PRC.

Description Added
9/21/00 4:38:41 PM

Virus Information

Discovery Date:	9/21/00
Origin:	IRC Chat Room
Length:	1,325 bytes
Type:	Virus
SubType:	PDA Device
Risk Assessment:	Low
Minimum Engine:	4.0.70
Minimum Dat:	4097
DAT Release Date:	09/27/2000

Virus Characteristics
McAfee AVERT discovered this virus Sept 21, 2000.

This is the **first virus designed for PalmOS**.

When an infected application is run, the screen is filled in dark gray box and then the program terminates. This virus will infect all third party applications on the PDA device. This virus overwrites the 1st section in the host .PRC file.

In testing, when a new program is copied to the Palm system via IR transfer, this program will execute normally. If another application which is already infected is run, the newly transferred file will then become infected.

Symptoms
Attempts to launch an application will result in the screen filled with a dark gray box pattern and then closes. The desired application fails to launch.

Method Of Infection

This virus will directly infect other PalmOS applications. Launching this program either accidentally or intentionally will result in the actions mentioned in the characteristics section of this description.

This virus copies its body to the 'code 1' resource of any other apps it finds. The original resource section is replaced with the virus code such that it is possible for infected applications to be smaller than the same program prior to infection.

Removal Instructions

Delete any file which contains this detection. Also delete phage.prc, if it exists, from your palm backup folder on your pc so you don't re-sync it back to your palm. This second step is necessary since the backup bit is set for phage.

Recovery from this threat requires a hard-reset followed by a hot-sync of the PDA device.

PalmOS ASCII Chart

20		21	!	22	"	23	#	24	$	25	%	26	&	27	'
28	(	29	)	2A	*	2B	+	2C	,	2D	-	2E	.	2F	/
30	0	31	1	32	2	33	3	34	4	35	5	36	6	37	7
38	8	39	9	3A	:	3B	;	3C	<	3D	=	3E	>	3F	?
40	@	41	A	42	B	43	C	44	D	45	E	46	F	47	G
48	H	49	I	4A	J	4B	K	4C	L	4D	M	4E	N	4F	O
50	P	51	Q	52	R	53	S	54	T	55	U	56	V	57	W
58	X	59	Y	5A	Z	5B	[	5C	\	5D	]	5E	^	5F	_
60	`	61	a	62	b	63	c	64	d	65	e	66	f	67	g
68	h	69	i	6A	j	6B	k	6C	l	6D	m	6E	n	6F	o
70	p	71	q	72	r	73	s	74	t	75	u	76	v	77	w
78	x	79	y	7A	z	7B	{	7C	|	7D	}	7E	~	7F	□
80	€	81	□	82	‚	83	ƒ	84	„	85	…	86	†	87	‡
88	ˆ	89	‰	8A	Š	8B	‹	8C	Œ	8D	♦	8E	♣	8F	♥
90	♠	91	‘	92	’	93	“	94	”	95	•	96	–	97	—
98	˜	99	™	9A	š	9B	›	9C	œ	9D	□	9E	□	9F	Ÿ
A0		A1	¡	A2	¢	A3	£	A4	¤	A5	¥	A6	¦	A7	§
A8	¨	A9	©	AA	ª	AB	«	AC	¬	AD	-	AE	®	AF	¯
B0	°	B1	±	B2	²	B3	³	B4	´	B5	µ	B6	¶	B7	·
B8	¸	B9	¹	BA	º	BB	»	BC	¼	BD	½	BE	¾	BF	¿
C0	À	C1	Á	C2	Â	C3	Ã	C4	Ä	C5	Å	C6	Æ	C7	Ç
C8	È	C9	É	CA	Ê	CB	Ë	CC	Ì	CD	Í	CE	Î	CF	Ï
D0	Ð	D1	Ñ	D2	Ò	D3	Ó	D4	Ô	D5	Õ	D6	Ö	D7	×
D8	Ø	D9	Ù	DA	Ú	DB	Û	DC	Ü	DD	Ý	DE	Þ	DF	ß
E0	à	E1	á	E2	â	E3	ã	E4	ä	E5	å	E6	æ	E7	ç
E8	è	E9	é	EA	ê	EB	ë	EC	ì	ED	í	EE	î	EF	ï
F0	ð	F1	ñ	F2	ò	F3	ó	F4	ô	F5	õ	F6	ö	F7	÷
F8	ø	F9	ù	FA	ú	FB	û	FC	ü	FD	ý	FE	þ	FF	ÿ

```
    struct WindowType * nextWindow;
  } WindowType;
```

Field Descriptions

`displayWidthV20` Width of the window in pre OS 3.5 devices. In OS 3.5, use `WinGetDisplayExtent` to return the window width.

`displayHeightV20` Height of the window in pre OS 3.5 devices. In OS 3.5, use `WinGetDisplayExtent` to return the window height.

`displayAddrV20` Pointer to the window display memory buffer in pre OS 3.5 devices. In OS 3.5 or later, call `WinGetBitmap` and then `BmpGetBits` to obtain the display's memory buffer.

`windowFlags` Window attributes (see `WindowFlagsType`).

`windowBounds` Display-relative bounds of the window. Use `WinGetWindowBounds` and `WinSetWindowBounds` to retrieve and set this value.

`clippingBounds` Bounds for clipping any drawing within the window. Use `WinGetClip` and `WinSetClip` to retrieve and set this value.

`bitmapP` Pointer to the window bitmap, which holds the window's contents. Use `WinGetBitmap` to retrieve this value.

`frameType` Frame attributes; see `FrameBitsType`.

`drawStateP` Pointer to a state of the current transfer mode, pattern mode, font, underline mode, and colors. See `DrawStateType`.
Only one drawing state exists in the system. Each window points to the same structure.

`nextWindow` Pointer to the next window in a linked list of windows. This linked list of windows is called the active window list.

```
/**
 * Compress a Bitmap (Tbmp or tAIB) resource.
 *
 * @param rcbmp      a reference to the Palm Computing resource data.
 * @param compress   compression style?
 * @param colortable does a color table need to be generated?
 */
static void
BMP_CompressBitmap(RCBITMAP *rcbmp,
                   int      compress,
                   BOOL     colortable)
{
  unsigned char *bits;
  int           size, msize, i, j, k, flag;

  // determine how much memory is required for compression (hopefully less)
  size  = 2 + (colortable? COLOR_TABLE_SIZE : 0);
  msize = size + ((rcbmp->cbRow + ((rcbmp->cbRow + 7) / 8)) * rcbmp->cy);

  // allocat memory and clear
  bits = (unsigned char *)malloc(msize * sizeof(unsigned char));
  memset(bits, 0, msize * sizeof(unsigned char));

  // do the compression (at least, attempt it)
  for (i=0; i<rcbmp->cy; i++) {
    flag = 0;
    for (j=0; j<rcbmp->cbRow; j++ ) {
      if ((i == 0) ||
          (rcbmp->pbBits[(i * rcbmp->cbRow) + j] !=
       rcbmp->pbBits[((i-1) * rcbmp->cbRow) + j])) {
        flag |= (0x80 >> (j & 7));
      }
      if (((j & 7) == 7) || (j == (rcbmp->cbRow-1))) {
        bits[size++] = (unsigned char) flag;
        for (k = (j & ~7); k <= j; ++k) {
          if (((flag <<= 1) & 0x100) != 0)
            bits[size++] = rcbmp->pbBits[i * rcbmp->cbRow + k];
        }
        flag = 0;
      }
    }
  }

  // if we must compress, or if it was worth it, save!
  if (compress == rwForceCompress || size < rcbmp->cbDst) {

    // do we have a color table?
    if (colortable) {

      int i;

      // copy the color table (dont forget it!)
      for (i=0; i<COLOR_TABLE_SIZE; i++)
        bits[i] = rcbmp->pbBits[i];

      bits[COLOR_TABLE_SIZE]    = (unsigned char)(size >> 8);
      bits[COLOR_TABLE_SIZE+1] = (unsigned char)size;
    }
    else {
      bits[0] = (unsigned char)((size & 0xff00) >> 8);
      bits[1] = (unsigned char) (size & 0x00ff);
    }

    // change the data chunk to the newly compressed data
    free(rcbmp->pbBits);
    rcbmp->ff     |= 0x8000;
    rcbmp->pbBits = bits;
    rcbmp->cbDst  = size;
  }
  else
    free(bits);
}
```

SECTION 4
LCD CONTROLLER MODULE

The liquid crystal display controller (LCDC) provides display data for an external LCD driver or LCD panel module. The key features include the following:

- Share system and display memory, no dedicated video memory required
- Standard panel interface for common LCD drivers
- Supports single (non-split) screen monochrome LCD panels
- Fast flyby type, 16-bit wide, burst-DMA screen refresh transfers from system memory
- Maximum display size is 1024x512; however, the typical non-split panel sizes are 320x240 and 640x200
- Panel interface: 1-, 2-, or 4-bit wide LCD data bus
- Black and white, or 4 simultaneous gray levels out of a palette of 7
- Hardware blinking cursor; programmable up to 32 x 32 pixels in size
- Hardware panning (soft horizontal scrolling)

The LCDC fetches display data directly from system memory through periodic DMA transfer cycles. The bus bandwidth used by the LCDC is low, thereby enabling the MC68EC000 core to have sufficient computing bandwidth for other tasks.

4.1 LCDC SYSTEM OVERVIEW

The LCDC is built of six basic blocks, namely MPU interface registers, screen DMA controller, line buffer, cursor logic, frame-rate control and LCD panel interface as shown in Figure 4-1.

4.1.1 MPU Interface

The MPU interface consists of all control registers that enable all different features of the LCDC. This block is connected directly to the 68K bus.

4.1.2 Direct Memory Access (DMA)

The DMA generates a bus-request signal to the MC68EC000 periodically and, upon receiving a bus grant, performs a 16- or 8-word memory burst to fill the line buffer. The number of DMA clock cycles per transfer is programmable (1, 2, 3, or 4 clocks/transfer), which makes it more versatile to support systems with memory of different speeds.

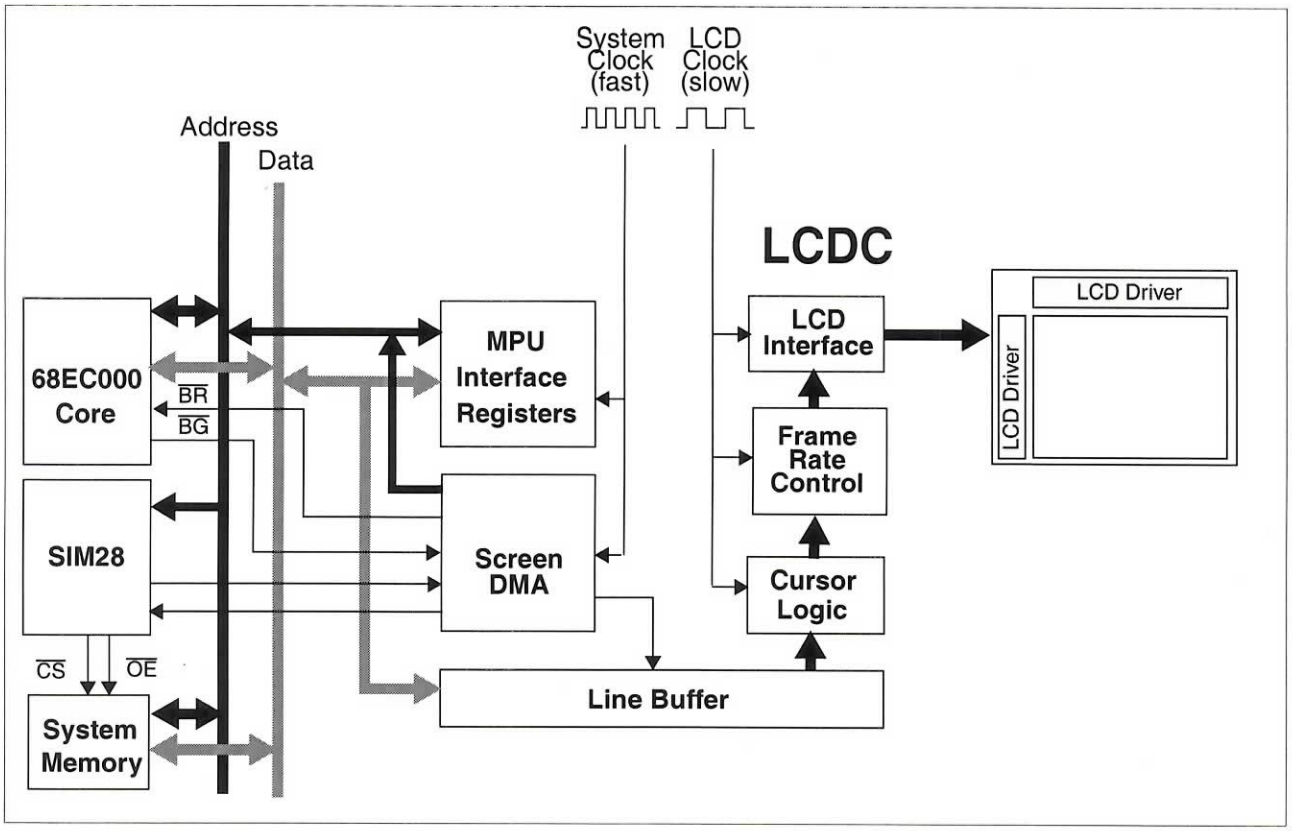


Figure 4-1. System Block Diagram of LCDC

4.1.3 Line Buffer

The line buffer collects display data from system memory during DMA cycles, and outputs it to the cursor-logic block. The input is synchronized with the fast DMA clock, while the output is synchronized to the relatively slow LCD pixel clock.

4.1.4 Cursor Control Logic

The cursor control logic (when enabled) generates a block-shaped cursor on the display screen. Users can adjust the cursor height and width to any number between 1 to 31. The cursor can be full black or reversed video, and the blinking rate is adjustable when the blink-enable bit is on.

4.1.5 Frame Rate Control (FRC)

The frame rate control (FRC) is used primarily for gray-scale display and can generate up to 4 gray levels from the choice of 7 density levels (0, 1/4, 5/16, 1/2, 11/16, 3/4, 1 as in Table 4-3). The density level corresponds to the number of times the pixel is being turned on when the display is refreshed frame by frame. Because the crystal formulations and driving voltage may vary, the visual gray quality can be tuned by programming the gray palette-mapping register (GPMR) to obtain the best effect.

Because blinking or flickering will occur if all LCD pixel cells are synchronized, it is essential to program two 4-bit numbers, namely Xoff and Yoff in the FRC offset register (FOSR), to minimize flickering. As a general rule, select odd numbers that differ by 2. The optimal offset values could vary among different models of LCD panels—even from the same manufacturer—because of different inter-pixel crosstalk characteristics.

4.1.6 LCD Interface

The LCD interface logic packs the display data in the correct size and outputs it to the LCD panel data bus. The polarity of FRM, LP, and SCLK signals as well as pixel data can all be programmable to suit different types of LCD panel requirements.

4.2 INTERFACING LCDC WITH LCD PANEL

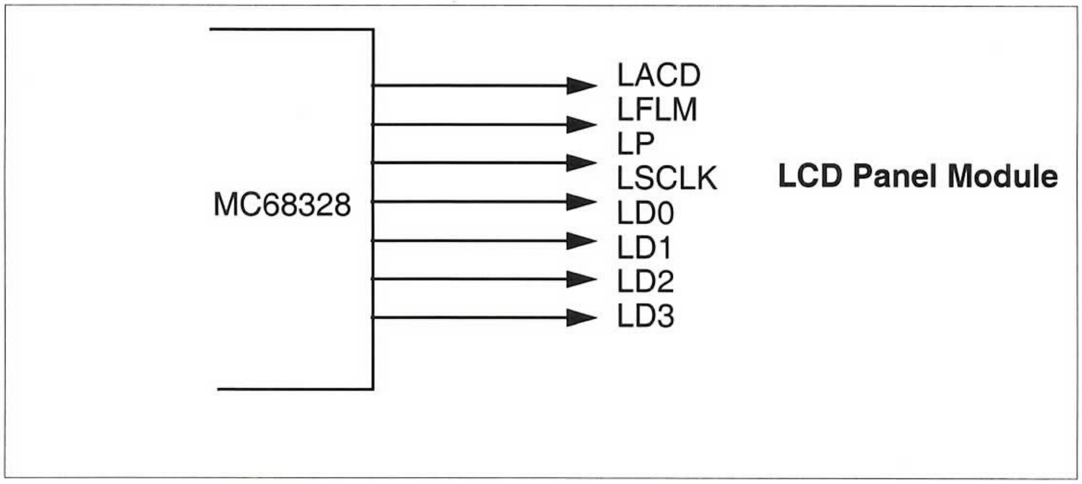


Figure 4-2. LCD Module Interface Signals

LCD Data Bus (LD3-LD0)

This output bus transfers pixel data to the LCD panel for display. Depending on which LCD panel mode was selected, data is arranged differently on the bus for each mode. Users can program the output pixel data to be negated. See the POLCF register description for details.

First Line Marker (LFLM)

This signal indicates the start of a new display frame. The LFLM signal becomes active after the first line pulse of the frame and remains active until the next line pulse, at which point it de-asserts and remains inactive until the next frame. Users can program the LFLM signal using software to be active-high or active-low. See the POLCF register description for details.

Line Pulse (LP)

This signal latches a line of shifted data onto the LCD panel. It becomes active when a line of pixel data is clocked into LCD panels and stays asserted for a duration of 8 pixel clock periods. Users can program the LP signal using software to be either active-high or active-low. See the POLCF register description for details.

Shift Clock (LSCLK)

This is the clock output that is synchronized to the LCD panel output data. Users can program the LSCLK signal using software to be either active-high or active-low. See the POLCF register description for details.

Alternate Crystal Direction (LACD)

This output is toggled to alternate the crystal polarization on the panel. Users can program this signal to toggle at a period of 1 to 16 frames. The alternate crystal direction (LACD, also called M) pin will toggle after a pre-programmed number of FLM pulses. Users can program the ACD rate-control register (ACDRC) so that LACD will toggle once every 1 to 16 frames. The targeted number of frames is equal to the alternation code's 4-bit value plus one. The default value for ACDRC is zero; that is, LACD will toggle on every frame. The LACD output signal is synchronized with the trailing (falling) edge of the line pulse (LP) enclosed by FLM.

Table 4-1. ACDRC Value and Number of Cycles

ACDRC	No of Cycles
0000	1
0001	2
0010	3
0100	5
1000	9
1111	16

4.3 PANEL I/F TIMING

The LCDC signal continuously pumps the pixel data into the LCD panel via the LCD data bus. The bus is timed by shift clock (LSCLK), line pulse (LLP) and first line marker (LFLM). The LSCLK clocks the pixel data into the display drivers' internal-shift register. The LP latches the shifted pixel data into a wide latch at the end of a line while the LFLM marks the first line of the displayed page.

The LCDC signal is designed for great flexibility to support most of the monochrome LCD panels available in the marketplace. Figure 4-3 shows the LCD interface timing for 8-bit LCD data-bus operations.

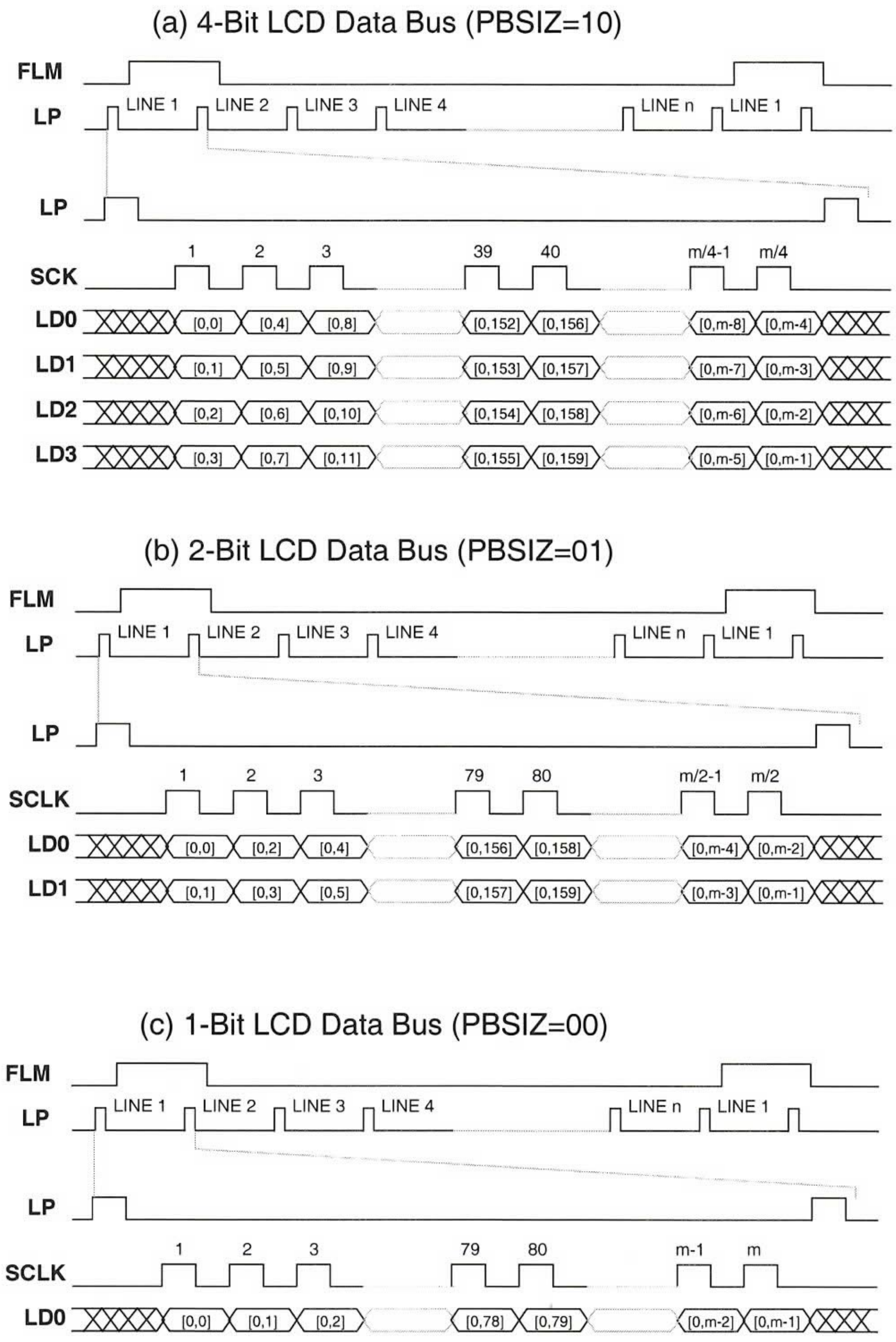
Figure 4-3 shows the LCD interface timing for 4-, 2-, and 1-bit LCD data-bus operations.

The line pulse signifies the end of the current line of serial data. The LLP enclosed by LFLM signal marks the end of the first line of the current frame.

Some LCD panels may use an active-low LFLM signal, LLP signal, LSCLK signal, and reversed pixel data. To change the polarities of these signals, set the first-line marker polarity (FLMPOL), line-pulse polarity (LPPOL), shift-clock polarity (SCLKPOL), and pixel polarity

(PIXPOL) bits to 1, respectively. The LLP and LFLM timing are similar for all panel modes supported by LCDC.

In additional to the interface timing pins discussed above, an alternate crystal direction (LACD) pin in LCDC will toggle after a pre-programmed number of LFLM pulses. This pin prevents crystal degradation in the LCD panel.



(a) 4-Bit LCD Data Bus (PBSIZ=10)

(b) 2-Bit LCD Data Bus (PBSIZ=01)

(c) 1-Bit LCD Data Bus (PBSIZ=00)

Figure 4-3. LCD Interface Timing for 4-, 2-, and 1-Bit Data Widths

4.4 OPERATION OVERVIEW

4.5 DISPLAY CONTROL

The LCDC signal drives single-screen monochrome STN LCD panels with up to 1024x512 pixels in the gray-scale mode at a refresh rate of 60-70 Hz. In any case, the best efficiency is achieved when the screen width is a multiple of the DMA controller's 16-bit bus width. Because of LCD driver-technology limitations, large screens, such as 640x480, are usually organized in spilt-screen format, which the MC68328 processor does not support. The actual limit is the number of rows that require high driving voltage. The MC68328 processor 4-bit LCD interface will drive up to 240 rows with a maximum of 1024 columns.

4.5.1 LCD Screen Format

The screen width and height of the LCD panel are software-programmable. Figure 4-4 shows the relationship between the portion of a large graphics file displayed on screen vs. the actual page. All units are measured in pixel counts in this figure.

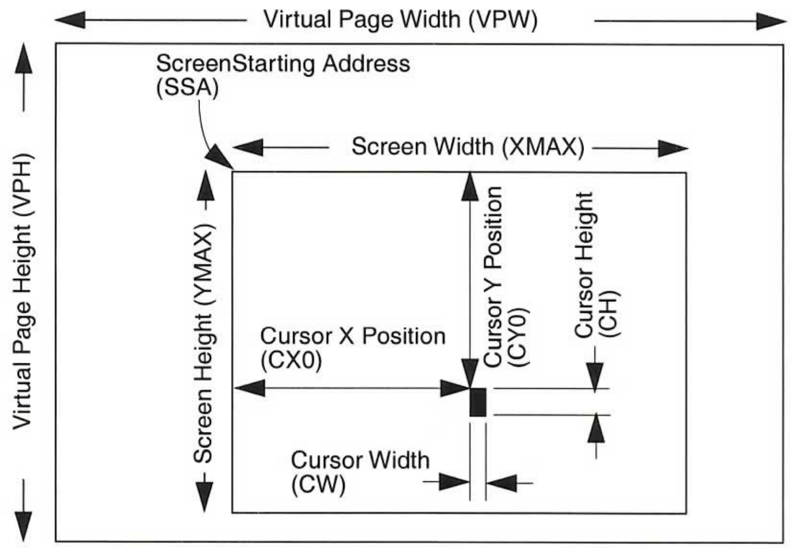


Figure 4-4. LCD Screen Format

The screen width (XMAX) and screen height (YMAX) registers specify the LCD panel size. The LCD will start scanning the display memory at the location pointed to by the screen starting address (SSA) register. Therefore, the LCD panel will display the shaded area in Figure 4-4.

The virtual page width (VPW) and virtual page height (VPH) parameters specify the maximum page width and height, respectively. By changing the screen-starting address (SSA) register, a screen-sized window can be vertically or horizontally scrolled (panned) anywhere inside the virtual-page boundaries. The software must position the starting address (SSA) properly so that the scanning logic's system memory pointer (SMP) does not stretch beyond VPW nor VPH. Otherwise, strange artifacts will display on the screen. The programmer uses the VPH only for boundary checks. There is no VPH register internal to the LCDC.

4.5.2 Cursor Control Logic

To define the position of the hardware cursor, the LCDC maintains a vertical line counter (YCNT) to keep track of the pixel's current vertical position. YCNT in conjunction with the horizontal pixel counter (XCNT) specify the screen position of the current pixel data being processed. When the pixel falls within a window specified by the cursor reference position (CXP:10-bit register, CYP: 9-bit register), cursor width (CW:5-bit counter), and cursor height (CH:5-bit counter), the original pixel bits (outputs of HSRA and B are affected but the latter's output is ignored, if not applicable) wil be passed transparently (cursor control bits=00), replaced with full black, or a complement for reversed video (CC bits=01,10 respectively; 11 not allowed) if a static cursor is chosen (BK_EN=0). Reversed video is preferable for a static cursor as it will block the original pixels if CC=01. A blinking cursor will display if BK_EN=1, in which case the original signal and the cursor will alternate periodically.

4.5.3 Display Data Mapping

The LCDC supports 1-bit-per-pixel or 2-bits-per-pixel graphics mode. In the binary mode (GS=0), each bit in the display memory corresponds to a pixel in the LCD panel. The corresponding pixel on the screen is either fully on or fully off.

In 2-bit-per-pixel operations (GS=1), the frame-rate control circuitry inside the LCDC will generate intermediate gray tones on the LCD panel by adjusting the densities of 1's and 0's over many frames. A maximum of 4 gray levels can be simultaneously displayed on the LCD screen.

The system memory data in both 1- and 2- bit-per-pixel modes are mapped as shown in Figure 4-5.

4.5.4 Gray Scale Generation

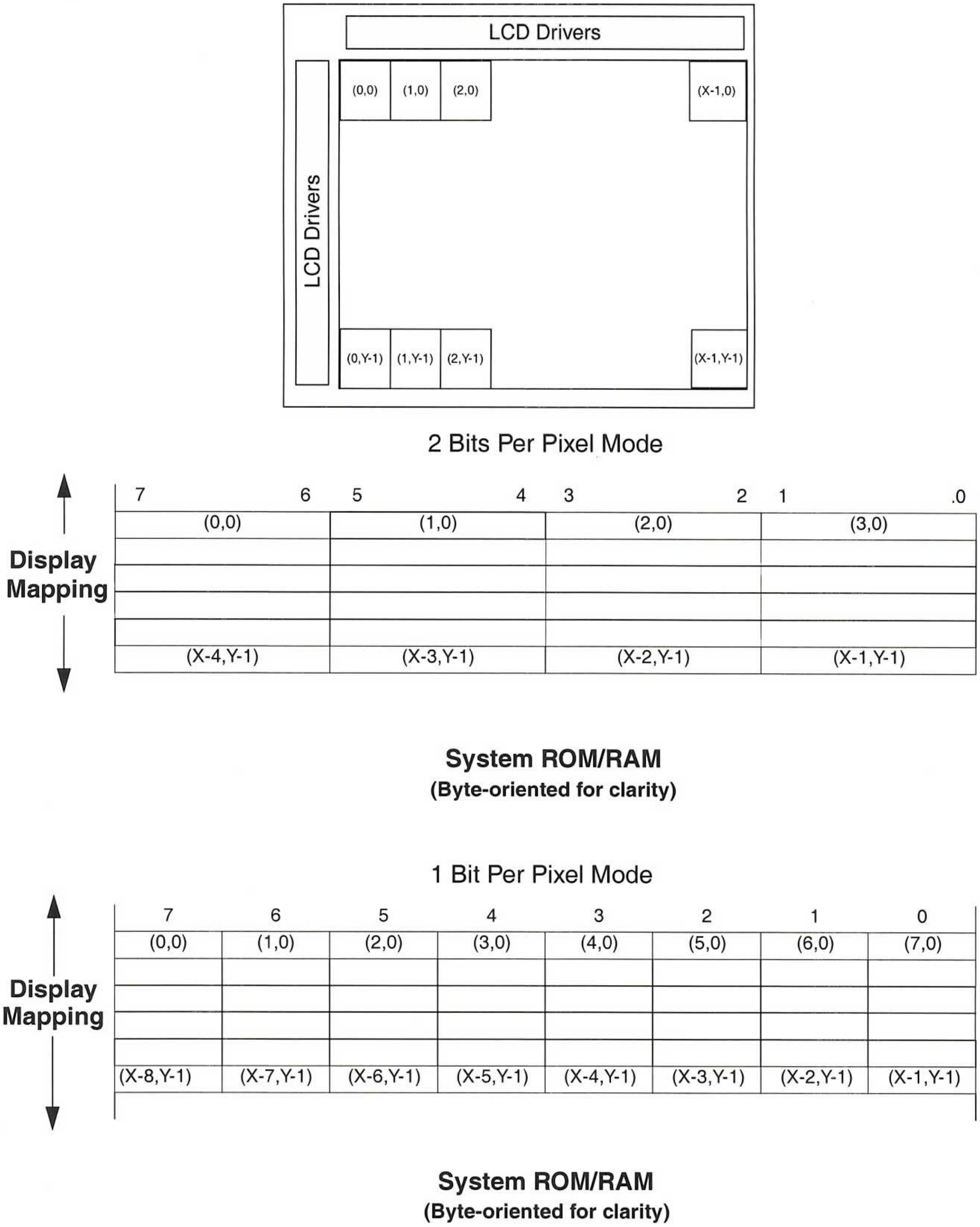
The LCDC is configured to drive only a single-screen monochrome LCD panel. It cannot handle color STN or TFT panels. Users can configure the data bus size for the LCD panel to 1-bit, 2-bit, or 4-bit by programming the LCD panel bus size (PBSIZ) register.

4.5.5 Gray Palette Mapping

Through a proprietary frame-rate control (FRC) algorithm, the LCDC can generate up to 4 simultaneous gray levels out of 7 available by first mapping the 2-bit data into four 3-bit gray codes which then select 4 out of 7 bit densities from the gray palette table.

Figure 4-5 shows the mapping of the 2-bit pixel data into 3-bit gray codes. Bits GMN are defined in the software-programmable gray palette mapping registers (GPMR). Each of the four 3-bit codes obtained from the first table then selects a density level (0, 1/4, 5/16, 1/2, 11/16, 3/4 and 1) from the gray palette table as shown in Table 4-3.

Because crystal formulations and driving voltages vary, the visual gray effect may or may not have a linear relationship to the frame rate. A logarithmic scale such as 0, 1/4, 1/2, and 1 might be more pleasing than a linear-spaced scale such as 0, 5/16, 11/16, and 1 for certain graphics. A flexible mapping scheme lets users optimize the visual effect for the specific panel or application.



2 Bits Per Pixel Mode

System ROM/RAM
(Byte-oriented for clarity)

1 Bit Per Pixel Mode

System ROM/RAM
(Byte-oriented for clarity)

Figure 4-5. Mapping of Memory Data on the Screen

Table 4-2. Gray Scale Code Mapping

Code Mapping	
Data	Gray code
00	G02G01G00
01	G12G11G10
10	G22G21G20
11	G32G31G30

Table 4-3. Gray Palette Selection

Gray Palette	
Gray Code	Density
000	0
001	1/4
010	5/16
011	1/2
100	11/16
101	3/4
110	1
111	1

4.5.6 FRC Offset Control

4.5.7 Cursor and Blinking Rate Control

4.5.8 Low-Power Mode

Some panels may have a signal called PANEL_OFF that turns off the panel for low-power mode. In the MC68328 processor system, this signal is not supported. Instead, use a parallel I/O pin to perform this function.

The software sequence to achieve PANEL_OFF using parallel I/O consists of 2 steps:

1. Turn off the VLCD (+15V or -15V) by I/O driving a transistor
2. Turn off the LCDON bit

To exit from LCDC-off mode:

1. Turn on the LCDON bit
2. Delay for 1-2ms
3. Turn on the VLCD by I/O driving a transistor

When setting the LCDON bit (register CKCON bit 7) to 1, LCDC itself will enter a low-power mode by stopping its own pixel clock prior to the next line-buffer-fill DMA. Additional screen DMA and display-refresh operations will then be stopped in this mode. When the LCDC is switched back on, DMA and screen-refresh activities will resume in a synchronous fashion. Software should check that the actual PANEL_OFF signal is de-asserted before setting LCDON to a 1.

4.6 DMA CONTROLLER OVERVIEW

The LCD DMA controller is a flyby type, 16-bit wide, fast-data transfer machine. Because the LCD screen has to be refreshed continuously at a rate of about 50-70 Hz, in this case, the pixel bits in the memory will be read and transferred to corresponding pixels on the screen. To minimize the bus obstruction because of bus-sharing with the system, a burst type and flyby transfer is therefore required. The refresh is divided into small packs of 8- or 16-word reads. Every time the internal line buffer needs data, it will assert the $\overline{BR}$ signal to request the bus from the MC68EC000. Once the MC68EC000 core grants the bus (i.e. $\overline{BG}$ is asserted), the DMA controller will get control of the bus signal and issue 8- or 16-word reads (see setting of CKCON register) from memory. The read data is then passed to the next stage internally to generate the LCD timing (flyby). During the LCD access cycles, output- enable and chip-select signals for the corresponding system SRAM chip will be asserted by the chip-select logic inside the SIM. The minimum bus bandwidth obstruct can be achieved by using zero LCD-access wait states (1 clock per access). See Section 4-8 Bandwidth Calculation and Saving for more details.

4.6.1 Basic Operation

As shown in Figure 4-6 and Figure 4-7, data is fetched from memory in a very efficient manner. Each burst is limited to 8/16 words, which reduces possible latency for other peripherals such as the interrupt controller. For example, the average time latency for LCDCLK = 5MHz with 16-word burst is approximately 2.4µs.

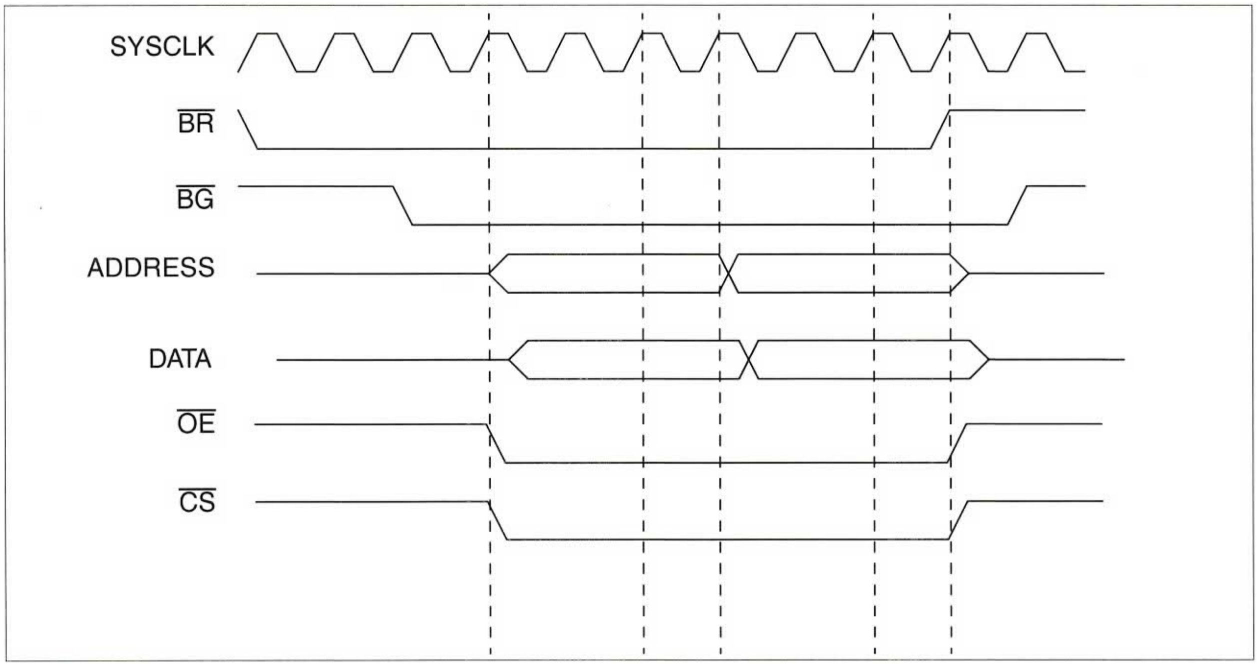


Figure 4-6. Three Clock per LCD DMA Transfer (2 Wait States)

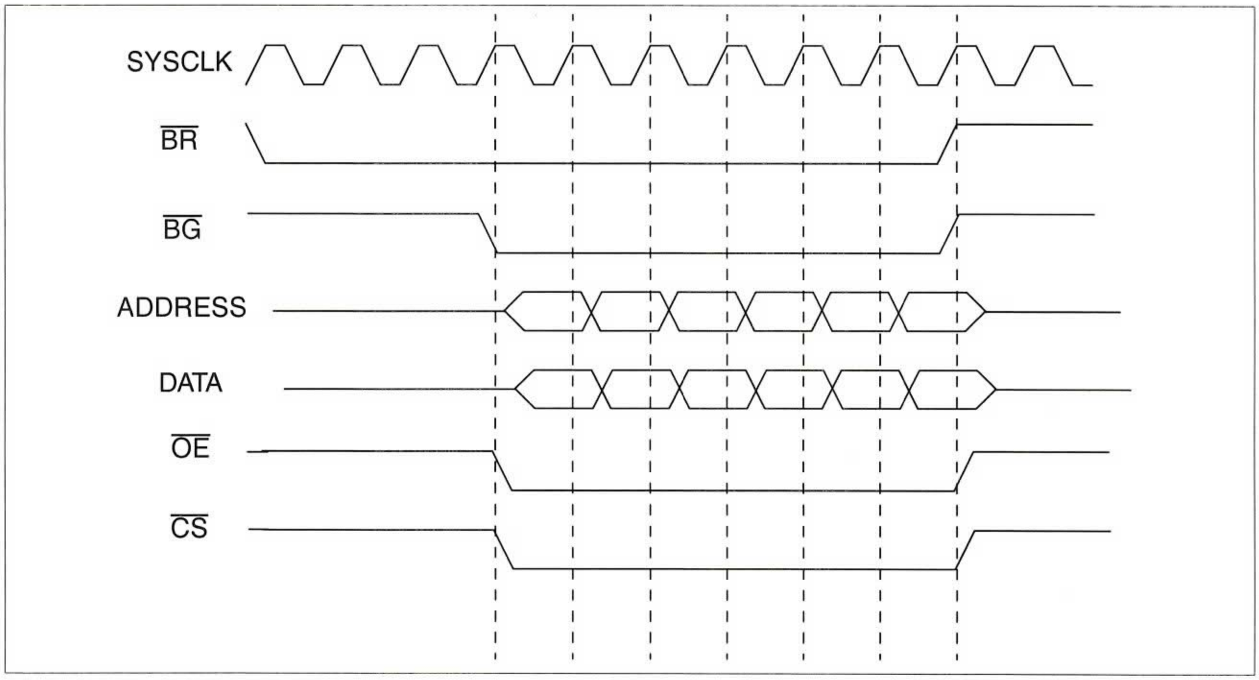


Figure 4-7. One Clock per DMA Transfer (0 Wait State)

4.7 REGISTER DESCRIPTIONS

4.7.1 System Memory Control Registers

4.7.1.1 SCREEN STARTING ADDRESS REGISTER (SSA).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SSA31	SSA30	SSA29	SSA28	SSA27	SSA26	SSA25	SSA24	SSA23	SSA22	SSA21	SSA20	SSA19	SSA18	SSA17	SSA16
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SSA15	SSA14	SSA13	SSA12	SSA11	SSA10	SSA9	SSA8	SSA7	SSA6	SSA5	SSA4	SSA3	SSA2	SSA1	0

Address: $(FF)FFFA00 Reset Value: $00000000

Figure 4-8. Screen Starting Address Register

SSA31-SSA1 Screen-Starting Address Register

32-bit screen-starting address of the LCD panel (see Figure 4-8). The LCDC fetches pixel data from system memory at this address.

4.7.1.2 VIRTUAL PAGE WIDTH REGISTER (VPW).

7	6	5	4	3	2	1	0
VP8	VP7	VP6	VP5	VP4	VP3	VP2	VP1

Address: $(FF)FFFA05 Reset Value: $FF

Figure 4-9. Virtual Page Width Register

VP8-VP0M Virtual Page Width Register

This register (see Figure 4-9) specifies the virtual page width of the LCD panel in terms of byte count. VP0 defaults to zero because of the 16-bit transfers.

VPW = virtual page width in pixels divided by c where c is 16 for black-and-white display and 8 for gray level.

4.7.2 Screen Format Registers

4.7.2.1 SCREEN WIDTH REGISTER (XMAX).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
UNUSED						XM9	XM8	XM7	XM6	XM5	XM4	XM3	XM2	XM1	XM0

Address: $(FF)FFFA08 Reset Value: $03FF

Figure 4-10. Screen Width Register XMAX

XM9-XM0

Pixels on a line are numbered 0 to XMAX for a screen width of XMAX +1 pixels. XMAX+1 must be a multiple of 16 (see Figure 4-10).

4.7.2.2 SCREEN HEIGHT REGISTER (YMAX).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
UNUSED						YM9	YM8	YM7	YM6	YM5	YM4	YM3	YM2	YM1	YM0

Address: $(FF)FFFA0A Reset Value: $01FF

Figure 4-11. Screen Height Register YMAX

YM8-YM0

This register (Figure 4-11) specifies the LCD panel height in term of pixels or lines. The lines are numbered from 0 to YMAX for a total of YMAX + 1 lines, which is equal to the screen height in pixel count.

4.7.3 Cursor Control Registers

4.7.3.1 CURSOR X POSITION REGISTER (CXP).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CC1	CC0	UNUSED				CXP9	CXP8	CXP7	CXP6	CXP5	CXP4	CXP3	CXP2	CXP1	CXP0

Address: $(FF)FFFA18 Reset Value: $0000

Figure 4-12. Cursor X Position Register

CC1-CC0

Cursor control bits

00= Transparent, cursor is disabled
01= Full density (black) cursor
10= Reversed video
11= Do not use

CXP9-CXP0

Cursor's horizontal starting position X in pixel count (from 0 to XMAX).

4.7.3.2 CURSOR Y POSITION REGISTER (CYP).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
UNUSED							CYP8	CYP7	CYP6	CYP5	CYP4	CYP3	CYP2	CYP1	CYP0

Address: $(FF)FFFA1A Reset Value: $0000

Figure 4-13. Cursor Y Position Register

CYP8-CYP0

Cursor's vertical starting position Y in pixel count (from 0 to YMAX).

4.7.3.3 CURSOR WIDTH & HEIGHT REGISTER (CWCH).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
UNUSED			CW4	CW3	CW2	CW1	CW0	UNUSED			CH4	CH3	CH2	CH1	CH0

Address: $(FF)FFFA1C Reset Value: $0101

Figure 4-14. Cursor Width & Height Register

CW4-CW0

Cursor width. This 5-bit group specifies the width of the hardware cursor in pixel count (from 1 to 31).

CH4-CH0

Cursor height. This 5-bit group specifies the height of the hardware cursor in pixel count (from 1 to 31).

NOTE

The cursor is disabled if either CW or CH are set to zero.

4.7.3.4 BLINK CONTROL REGISTER (BLKC).

7	6	5	4	3	2	1	0
BKEN	BD6	BD5	BD4	BD3	BD2	BD1	BD0

Address: $(FF)FFFA1F Reset Value: $7F

Figure 4-15. Blink Control Register

BKEN

Blink-enable cursor will remain on instead of blinking if this bit is cleared. Defaults to zero.

1 = Blink enable
0 = Blink disable

BD6-BD0

Blink divisor. The cursor will toggle once per specified number of internal frame pulses plus one. The half-period may be as long as 2 seconds.

4.7.4 LCD Panel Interface Registers

4.7.4.1 PANEL INTERFACE CONFIGURATION REGISTER (PICF).

7	6	5	4	3	2	1	0
UNUSED					PBSIZ1	PBSIZ0	GS

Address: $(FF)FFFA20 Reset Value: $00

Figure 4-16. Panel Interface Configuration Register

PBSIZ1-PBSIZ0 Panel Bus Width

LCD panel bus size.

00 = 1-bit
01 = 2-bit
10 = 4-bit
11 = unused

GS Gray Scale

Gray scale mode bit. This bit, if set, enables 4-gray level (2 bits per pixel) mode. Its default value is 0, which selects binary pixel (no gray scale) operation.

1 = Gray scale enable
2 = No gray scale

4.7.4.2 POLARITY CONFIGURATION REGISTER (POLCF).

7	6	5	4	3	2	1	0
UNUSED				LCKPOL	FLMPOL	LPPOL	PIXPOL

Address: $(FF)FFFA21 Reset Value: $00

Figure 4-17. Polarity Configuration Register

LCKPOL LCD Shift Clock Polarity

This bit controls the polarity of the LCD shift-clock active edge.

0 = Active negative edge of LCLK
1 = Active positive edge of LCLK

FLMPOL

First-line marker polarity

0 = Active High
1 = Active Low

LPPOL

Line-pulse polarity

0 = Active-high
1 = Active-low

PIXPOL

Pixel polarity

0 = Active-high
1 = Active-low

4.7.4.3 LACD (M) RATE CONTROL REGISTER (ACDRC).

7	6	5	4	3	2	1	0
UNUSED				ACD3	ACD2	ACD1	ACD0

Address: $(FF)FFFA23 Reset Value: $00

Figure 4-18. LACD Rate Control Register

ACD3-ACD0 Alternate Crystal Direction Control

ACD toggle-rate control code. The ACD signal will toggle once every 1 to 16 FLM cycles based on the value specified in ACDRC register. The actual number of FLM cycles is the value programmed plus one. Shorter cycles tend to give better results.

4.7.5 Line Buffer Control Registers

4.7.5.1 PIXEL CLOCK DIVIDER REGISTER (PXCD).

7	6	5	4	3	2	1	0
UNUSED		PCD5	PCD4	PCD3	PCD2	PCD1	PCD0

Address: $(FF)FFFA25 Reset Value: $00

Figure 4-19. Pixel Clock Divider Register

PCD5-PCD0 Pixel Clock Divider

The PIX clock from the PLL is divided by N (PCD5-0 plus one) to yield the actual pixel clock. Values of 1-63 will yield N=2 to 64. If set to 0 (N=1), the PIX clock will be used directly, bypassing the divider circuit. Input source is selected by PCDS in CKCON register.

4.7.5.2 CLOCKING CONTROL REGISTER (CKCON).

7	6	5	4	3	2	1	0
LCDON	DMA16	WS1	WS0	UNUSED		DWIDTH	PCDS

Address: $(FF)FFFA27 Reset Value: $00

Figure 4-20. Clocking Control Register

LCDCON

This bit controls the LCDC block.

0 = Disable LCDC
1 = Enable LCDC

NOTE

The internal LCDC logic will be switched off in step with the FLM pulse.

DMA16

This bit controls the length of the DMA burst.

0 = 8 words burst length
1 = 16 words burst length

WS1-WS0 DMA Bursting Clock Control

Number of clock cycles per DMA word access

00 = Single clock-cycle transfer
01 = Two clock-cycle transfer
10 = Three clock-cycle transfer
11 = Four clock-cycle transfer

DWIDTH

Displays memory-data width indicating the size of the external bus interface.

0 = 16-bits memory
1 = 8-bits memory

PCDS Pixel Clock Divider Source Select

0 = The SYS CLK output of PLL is selected
1 = The PIX CLK output of PLL is selected

4.7.5.3 LAST BUFFER ADDRESS REGISTER (LBAR).

7	6	5	4	3	2	1	0
UNUSED	LBAR7	LBAR6	LBAR5	LBAR4	LBAR3	LBAR2	LBAR1

Address: $(FF)FFFA29 Reset Value: $3E

Figure 4-21. Last Buffer Address Register

LBA7-LBA1

The number of memory words required to fill one line on the display panel. The count is typically equal to the screen width in pixels divided by 16 for black-and-white display, or by 8 if in gray scale. For panning, add one more count for black-and-white and two for gray display.

4.7.5.4 OCTET TERMINAL COUNT REGISTER(OTCR).

7	6	5	4	3	2	1	0
OTC8	OTC7	OCT6	OCT5	OTC4	OTC3	OTC2	OTC1

Address: $(FF)FFFA2B Reset Value: $3F

Figure 4-22. Octet Terminal Count Register

OTC8-OTC1

Controls the time interval between two lines; therefore, the frame refresh rate can also be finely adjusted. The register value must be greater than LBAR by 4 for black-and-white display and 8 for gray display.

4.7.5.5 PANNING OFFSET REGISTER (POSR).

7	6	5	4	3	2	1	0
UNUSED				BOS	POS2	POS1	POS0

Address: $(FF)FFFA2D Reset Value: $00

Figure 4-23. Panning Offset Register

BOS Byte Offset

BOS is used primarily in the non-gray scale mode and in conjunction with POS0-2. (BOS must be set to zero for gray-scale data).

0 = Start from the first byte when retrieving binary pixel data for display
1 = Active display will start from the second byte instead

NOTE

The cursor reference position must be adjusted separately with software when this register is changed.

POS2-POS0 Pixel Offset Code

POS specifies which of the 8 pixels in the first or second (GS=0, BOS=1 only) octet retrieved from the line buffer is the first to be displayed on the screen. (e.g. 000 implies that pixel 7, the first shifted out, will be the first to be displayed on every horizontal line in the current frame).

4.7.6 Gray-Scale Control Registers

4.7.6.1 FRAME-RATE MODULATION CONTROL REGISTER (FRCM) .

7	6	5	4	3	2	1	0
XMOD3-XMOD0				YMOD3-YMOD0			

Address: $(FF)FFFA31 Reset Value: $B9

Figure 4-24. Frame-Rate Modulation Control Register

XMOD, YMOD Frame-Rate Modulation Control

These numbers modulate adjacent pixels at different time periods to avoid spatial flicker or jitter when using FRC. These values must be optimized by manually fine tuning the target LCD panel. See **Section 4.5.5 Gray Palette Mapping** for details.

4.7.6.2 GRAY PALETTE MAPPING REGISTER (GPMR).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	G12	G11	G10	0	G02	G01	G00	0	G32	G31	G30	0	G22	G21	G20

Address: $(FF)FFFA32 Reset Value: $0173

Figure 4-25. Gray Palette Mapping Register

GMN

Gray palette code (bit position n=0, 1, 2) output for pixel-input data m (0 for pixel data 00, 1=01, 2=10, 3=11). This 3-bit code will then select one of 7 bitstreams of different densities. See **Section 4.5.5 Gray Palette Mapping** for details.

4.8 BANDWIDTH CALCULATION AND SAVING

Because LCD screen refresh is a periodic task, the load LCDC puts on the host data bus becomes an important consideration to the high-performance handheld system designer.

4.8.1 Bus Overhead Considerations

The following example illustrates the issues involved in the estimation of bandwidth overhead to the data bus.

Consider a typical case scenario:

Screen size: 320 x 240 pixels

Bits per pixel: 2 bits / pixel

Screen refresh rate: 60 Hz

System clock = 16.67 MHz

Host bus size: 16 bit

DMA access cycle: 2 cycles per 16-bit word

The period, T_l,, that LCDC must update one line of the screen is,

$$\begin{aligned} T_l &= \frac{1}{60\mathrm{Hz}} \times \frac{1}{240\mathrm{lines}} \\ &= 69.4\mu s \end{aligned} \quad \text{(EQ 1)}$$

At the same period, the line buffer must be filled. The duration, T_{DMA}, which the DMA cycle will take up the bus is,

$$\begin{aligned} T_{DMA} &= \frac{320\mathrm{pixels} \times 2bitperpixel \times 2clock}{16.67MHz \times 16bitbus} \\ &= 4.8\mu s \end{aligned} \quad \text{(EQ 2)}$$

Thus, the percentage of host bus time taken up by the LCDC DMA is P_{DMA},

$$\begin{aligned} P_{DMA} &= \frac{4.8\ \mu s}{69.4\ \mu s} \\ &= 6.92\% \end{aligned} \quad \text{(EQ 3)}$$

3. 16-Bit Color Support

Applications using the Palm OS 3.5 color APIs will work on Visor Prism without modification. A color depth of up to 8 bits per pixel is supported through an index mechanism. A full description of the associated functions and data structures is documented in the Palm OS 3.5 development guides.

An index is used to reference an 8-bit pixel value to an RGB color that that will be displayed on the LCD. By contrast, 16-bit color support is implemented using a DirectColor mechanism. A 16-bit RGB value is used directly, rather than referenced through an index table, to provide pixel color information. Developers that wish to take advantage of the DirectColor will need to use the new API calls described in this document.

3.1 Determining if 16-bit color APIs are present

The API calls documented here are only necessary for applications that wish to take full advantage of the expanded range of colors available with a direct color display. Applications written to use the base OS 3.5 color API calls will continue to work without modification on systems with direct color displays.

To determine if the new API calls for direct color are available, an application should check to see if the `WinSetForeColorRGB()` system call is implemented. If this call is available, then *all* of the direct color API calls documented here are available:

```
if (SysGetTrapAddress (sysTrapWinSetForeColorRGB)

       != SysGetTrapAddress (sysTrapSysUnimplemented))

  directColorAvailable = true;

else

  directColorAvailable = false;
```

3.2 Foreground, Background, and Text colors

The base Palm OS 3.5 API calls for setting the foreground, background, and text colors are designed to take an index value as the color parameter. The index value is basically an index into a color lookup table where each entry in the table specifies the red, green, and blue components of the color. In order to set a certain drawing color for example, an application first has to lookup the index of that color in the color lookup table (using `WinRGBToIndex`) before passing that index value to `WinSetForeColor()`.

Displays that support 1, 2, 4, or 8 bits per pixel rely on a color lookup table in the display hardware in order to map pixel values into actual colors and the only colors that can be displayed on the screen at any given time are those that are found in the display's color lookup table. Thus, the indexed form of the set color API calls covers the entire range of available colors.

Direct color displays on the other hand, do not rely on a color lookup table because the value stored into each pixel location specifies the amount of red, green, and blue components directly. For example, a 16-bit direct color display could have 5 bits of each pixel assigned as the red component, 6 bits as the green component, and 5 bits as the blue component. With this type of display, the application is no longer limited to drawing with a color that is in a color lookup table.

The base indexed mode calls for setting the foreground, background, and text colors continue to work even with direct color displays because *the system uses a translation table for mapping color index values into actual colors.* This translation table is the color lookup table of the destination bitmap. If the destination bitmap does not include a color lookup table (the more common case), then the color lookup table of the screen itself is used. The screen's color lookup table contains the system's default palette of colors, but it can be changed through the `WinPalette()` call. When screen is configured as a direct color display, the system gives it an 8-bit color table, providing 256 possible indexed colors (the maximum possible for indexed modes). Note that this color lookup table for the screen is present only for compatibility with the indexed mode color calls and has no effect on the actual display hardware itself since the display hardware derives the color from the actual red, green, and blue bits stored in each pixel location of the frame buffer.

Even though the indexed mode calls continue to work with direct color displays, new forms of the calls must be introduced to make the entire range of direct colors available to an application. The new forms of these calls take an `RGBColorType` parameter that specifies the exact amount of red, green, and blue to use and thus are not limited to colors in the destination bitmap's color lookup table.

The prototypes of these calls are shown below. For each call, `newRgbP` is the new color to use and the previous color is returned *in* `prevRgbP`. These new calls are more generic than the older indexed forms and can be used with both indexed (1, 2, 4, or 8 bit) or direct color displays (the system will automatically look up the color index value of the closest color for you, if necessary).

```
WinSetForeColorRGB (const RGBColorType* newRgbP,
                          RGBColorType* prevRgbP);
WinSetBackColorRGB (const RGBColorType* newRgbP,
                          RGBColorType* prevRgbP);
WinSetTextColorRGB (const RGBColorType* newRgbP,
                          RGBColorType* prevRgbP);
```

Because these new RGB forms of the calls are only available on systems with the direct color enhancements present, applications should generally stick to using the older indexed form of these calls (`WinSetForeColor()`, `WinSetBackColor()`, `WinSetTextColor()`) unless they need finer control over the choice and dynamic range of colors.

3.3 Pixel Reading and Writing

The base OS 3.5 API call for reading a pixel value, `WinGetPixel()`, is designed to return a color index value (`IndexedColorType`). When this call is performed on a direct color display, it must first get the actual pixel value (a 16 or 24 bit direct color value) and then look up the closest color from the bitmap's color lookup table and return the index of the closest color from that table. If the bitmap does not include a color table (the common case), then the 256 entry color table for the screen itself is referenced. This mode of operation ensures compatibility for applications that will in turn take the return value from `WinGetPixel()` and use it as an indexed color to `WinSetForeColor()`, `WinSetBackColor()`, etc.

If the intent of the application is to copy pixels exactly from one bitmap to another, the application will experience a loss of color accuracy on a direct color display because of the closest-match color table lookup operation that `WinGetPixel()` performs.

To avoid this potential loss of color accuracy with direct color displays, applications can instead use the new `WinGetPixelRGB()` call. This call returns the pixel as an `RGBColorType` with a full 8 bits each of red, green, and blue ensuring no loss of color resolution. This new call is more generic than the base `WinGetPixel()` call, and can be used with both indexed (1, 2, 4, or 8 bit) or direct color displays (the system will automatically look up the RGB components of indexed color pixels as necessary). The prototype of this function is:

```
WinGetPixelRGB (Coord x, Coord y, RGBColorType* rgbP);
```

The pixel setting API calls (`WinPaintPixel()`, `WinDrawPixel()`, etc.) all rely on using the current foreground and background colors and don't require new forms for direct color displays. An application can simply pass in the return `RGBColorType` from `WinGetPixelRGB()` to `WinSetForeColorRGB()` and then call `WinDrawPixel()` in order to copy a direct color pixel.

3.4 Direct Color Bitmaps

The new Window and Blitter managers now support 16-bits per pixel direct color bitmaps, as well as the previously supported 1-, 2-, 4-, and 8-bit indexed color bitmaps. The format and version of the BitmapType structure is still 2, but a new directColor flag is defined for the flags field. This bit, when set, indicates that the pixels in the bitmap are direct color pixels. In addition to this flag, a direct color bitmap must also include the following four fields. If the bitmap also includes a color lookup table, then these fields follow the color lookup table; otherwise they immediately follow the bitmap structure itself.

```
typedef struct BitmapDirectInfoType
  {
   UInt8          redBits;
   UInt8          greenBits;
   UInt8          blueBits;
   UInt8          reserved;     // <- must be zero
   RGBColorType     transparentColor;
  }
BitmapDirectInfoType;
```

The redBits, greenBits, and blueBits fields indicate the number of bits in each pixel for each color component. The current implementation only supports 16-bits per pixel bitmaps with 5 bits of red, 6 bits of green, and 5 bits of blue. This type of 16-bit direct color pixel is laid out like this:

```
R R R R  R G G G  G G G B  B B B B
MSB                              LSB
```

The transparentColor field contains the red, green, and blue components of the transparent color of the bitmap. For direct color bitmaps, this field is used instead of the transparentIndex field to designate the transparent color value of the bitmap, since the transparentIndex field is only 8 bits wide and can only represent an indexed color. The transparentColor field, like the transparentIndex field, is ignored unless the hasTransparency bit is set in the bitmap's flags field.

It is important to note that as long as the new version 3 Window manager is present, a 16-bit direct color bitmap can always be rendered, *regardless* of the actual screen depth. The color APIs will automatically perform the necessary bit depth conversion to render the bitmap into whatever the depth of the destination.

Bitmap resources can be built to contain multiple depth images in the same bitmap resource - up to one per each possible depth. A potential incompatibility could arise if an application includes only a direct color version of a bitmap, though. Trying to draw a direct color bitmap with an older version of the Blitter manager will cause the system to crash. Consequently, applications need to either check that version 3 of the Window manager is present before drawing a direct color bitmap, or they must always include a 1, 2, 4, or 8 bit per pixel image of the bitmap in the bitmap resource along with the direct color version.

3.5 Special Drawing Modes

The special drawing modes of winErase, winMask, winInvert, and winOverlay introduce a complication when it comes to direct color models. These drawing modes were originally conceived of for use with monochrome bitmaps in which black is designated by 1 bits and white is designated by 0 bits. With these "color" assignments, these various modes can be described as:

- WinErase becomes an AND operation (black pixels in the source leave the destination alone, whereas white pixels in the source make the destination white).

- `WinMask` becomes an ANDNOT operation (black pixels in the source make the destination white, whereas white pixels leave the destination alone).
- `WinInvert` becomes an XOR operation (black pixels in the source invert the destination, whereas white pixels leave the destination alone).
- `WinOverlay` becomes an OR operation (black pixels in the source make the destination black, whereas white pixels in the source leave the destination alone).

In a direct color bitmap, black is designated by all 0's and white is designated by all 1's. Because of this, if all the drawing modes were implemented as logical operations in the same way they were for indexed color modes, the desired *effect* would not be achieved.

The assumption made by the direct color APIs is that the desired effect is more important to the caller than the actual logical operation that is performed. Thus, the various drawing modes, when drawing to a direct color bitmap, become:

- `WinErase` becomes an OR operation (black pixels in the source leave the destination alone, whereas white pixels in the source make the destination white).
- `WinMask` becomes an ORNOT operation (black pixels in the source make the destination white, whereas white pixels leave the destination alone).
- `WinInvert` becomes an XORNOT operation (black pixels in the source invert the destination, whereas white pixels leave the destination alone).
- `WinOverlay` becomes an AND operation (black pixels in the source make the destination black, whereas white pixels in the source leave the destination alone).

As long as the source and destination bitmaps contain only black and white colors, the new interpretations of the drawing modes in direct color modes will produce the same effects as they would with an indexed color mode. With non-black-and-white pixels, however, an application may get unexpected results from these drawing modes if they assume that the color APIs will perform the same logical operation in direct color mode as they do in indexed color mode.

SECTION 12
LCD CONTROLLER

The liquid crystal display (LCD) controller provides display data for external LCD drivers or for an LCD panel. The LCD controller fetches display data directly from system memory through periodic DMA transfer cycles. It uses very little bus bandwidth, which gives the core sufficient processing time. The following list contains the features of the LCD controller.

- Shares system and display memory, but no dedicated video memory is required
- Standard panel interface for common LCD drivers
- Supports single (non-split) screen monochrome/color STN LCD panels
- Fast fly-by type, 16-bit wide burst DMA screen refresh transfers from system memory
- Maximum display size is 640x512 pixels for b/w and 320x240 for gray display
- Panel interface of 4-, 2-, and 1-bit wide LCD data bus
- Four or sixteen simultaneous gray-scale levels from a palette of 16
- Hardware blinking cursor that is programmable at a maximum 31x31 pixels
- Hardware panning (soft horizontal scrolling)
- 8-bit pulse-width modulator for software contrast control

The LCD controller consists of MPU interface registers, control logic, a screen DMA controller, line buffer, cursor logic, frame rate control, and an LCD panel interface. Figure 12-1 illustrates how these blocks are organized in the LCD controller.

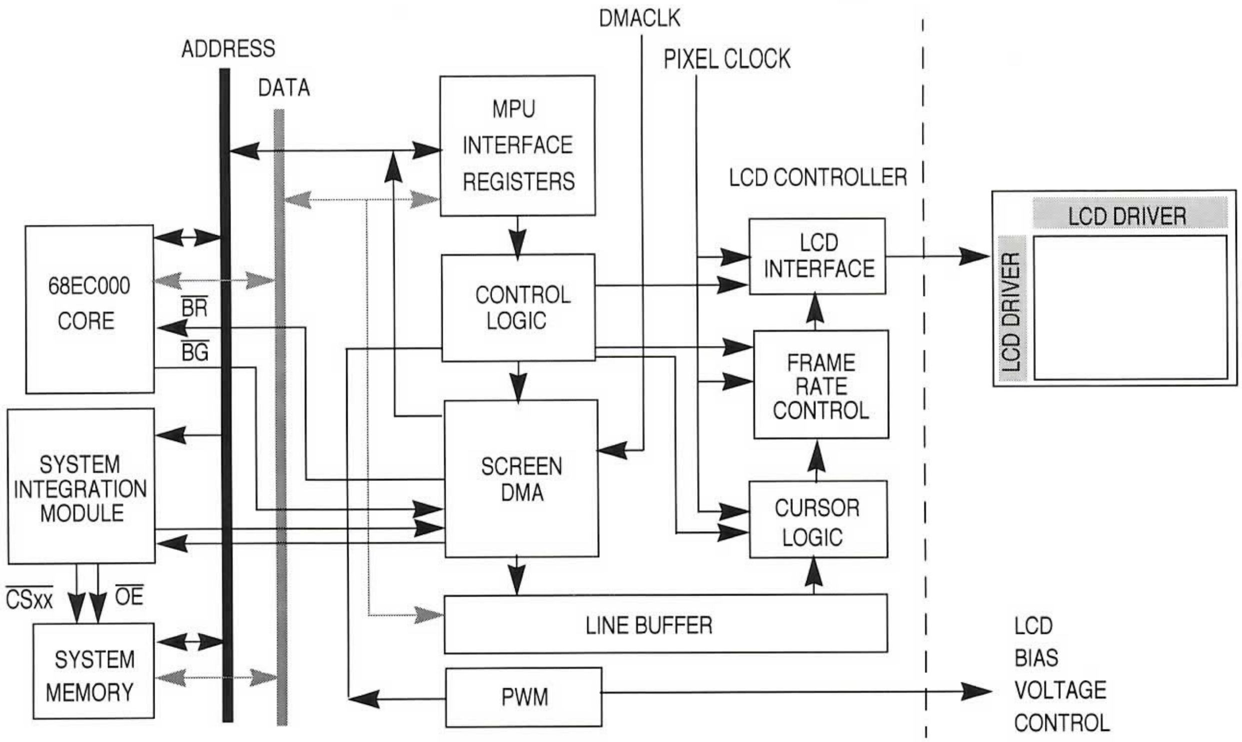


Figure 12-1. LCD Controller Block Diagram

12.1 OPERATION

The MPU interface registers enable the different features of the LCD controller. They are connected to the 68K bus. The control logic provides the internal control and counting signals for other blocks. The DMA generates a bus request (BR) signal to the core and when the bus is granted, it performs a few memory bursts to fill up the line buffer. The number of DMA clock cycles in each burst is the programmable number of clocks per transfer, which makes it easier to support a system with memory that has different speed grades.

The line buffer collects display data from system memory during DMA cycles and outputs it to the cursor logic block. The input is synchronized with the fast DMA clock, while the output is synchronized to the relatively slow LCD pixel clock. The cursor control logic, when enabled, is used to generate a block-shaped cursor on the display screen. You can change the height and width of the cursor, as long as you use a number between 1 and 31. The cursor can be completely black or reversed video and the blinking rate is adjustable when the BKEN bit in the LCD blink control (LBLKC) register is set.

Frame rate control is mainly used for gray-scale displays and can generate a maximum of sixteen gray-scale levels out of 16 density levels, as shown in Table 12-1. The density level corresponds to the number of times that a pixel is turned on when the display is refreshing. Since crystal formulations and driving voltage may vary, the quality of the gray-scale can be fine-tuned by programming the LCD gray palette mapping register (LGPMR).

The LCD interface logic is used to pack the display data into the correct size and output it to the LCD panel's data bus. The polarity of the LFLM, LP, and LCLK signals and pixel data can all be programmed to suit different LCD panel requirements.

12.1.1 Connecting the LCD Controller to an LCD Panel

The following signals are used to connect the LCD controller to an LCD panel:

- **LD[3:0].** The LCD Data Bus lines transfer pixel data to the LCD panel so that it can be displayed. Depending on which LCD panel mode is selected, data is arranged differently on the bus. You can program the output pixel data to be negated. See Section 12.2.10 LCD Polarity Configuration Register for more information. The LCD controller is configured to drive single-screen monochrome LCD panels only. The data bus size for an LCD panel can be configured to 1-, 2-, or 4-bit by programming the LCD panel bus size (LPBSIZ) register.
- **LFLM.** The LCD Frame Marker signal indicates the start of a new display frame. LFLM becomes active after the first line pulse of the frame and remains active until the next line pulse, at which point it deasserts and remains inactive until the next frame. You can program LFLM to be an active high or active low signal in software. See Section 12.2.10 LCD Polarity Configuration Register for more information.
- **LLP.** The LCD Line Pulse signal is used to latch a line of shifted data onto an LCD panel. It becomes active when a line of pixel data is clocked into the LCD panel and stays asserted for an 8-pixel clock period. You can program LLP to be an active high or active low signal in software. See Section 12.2.10 LCD Polarity Configuration Register for more information.
- **LCLK.** The LCD Shift Clock signal is the clock output to which the output data to the LCD panel is synchronized. You can program LCLK to be an active high or active low signal in software. See Section 12.2.10 LCD Polarity Configuration Register for more information.
- **LACD.** The LCD Alternate Crystal Direction output signal is toggled to alternate the crystal polarization on the panel. You can program this signal to toggle for a period of 1 to 16 frames. The LACD signal will toggle after a preprogrammed number of FLM or LP pulses. The LACD rate control register (LACDRC) can be programmed so that LACD will toggle once every 1 to 16 frames. The targeted number is equal to the alternation code's 4-bit value plus one. The default value for LACDRC is zero, which enables the LACD signal to toggle on every frame. The LACD signal is synchronized with the trailing (falling) edge of the LLP signal, which is enclosed by the LFLM signal. See Section 12.2.11 LACD Rate Control Register for more information.

12.1.1.1 PANEL INTERFACE TIMING. The LCD controller continuously pumps the pixel data into the LCD panel via the LCD data bus. The bus is timed by the LCLK, LLP and LFLM signals. The LCLK signal clocks the pixel data into the display drivers' internal shift register. The LLP signal latches the shifted pixel data into a wide latch at the end of a line while the LFLM signal marks the first line of the displayed page.

The LCD controller is designed to support most monochrome LCD panels. Figure 12-2 illustrates the LCD interface timing for 1-, 2-, and 4-bit LCD data bus operation. The LLP signal signifies the end of the current line of serial data. The LLP signal enclosed by the LFLM signal marks the end of the first line of the current frame.

Some LCD panels can use an active low LFLM, LLP, or LCLK signal and reversed pixel data. To change the polarity of these signals, set the FLMPOL, LPPOL, LCKPOL and PIXPOL bits in the LCD polarity configuration (LPOLCF) register to 1. In addition to the interface timing pins, the LACD pin will toggle after a preprogrammed number of LFLM pulses. The purpose of this pin is to prevent the crystal in the LCD panel from degrading.

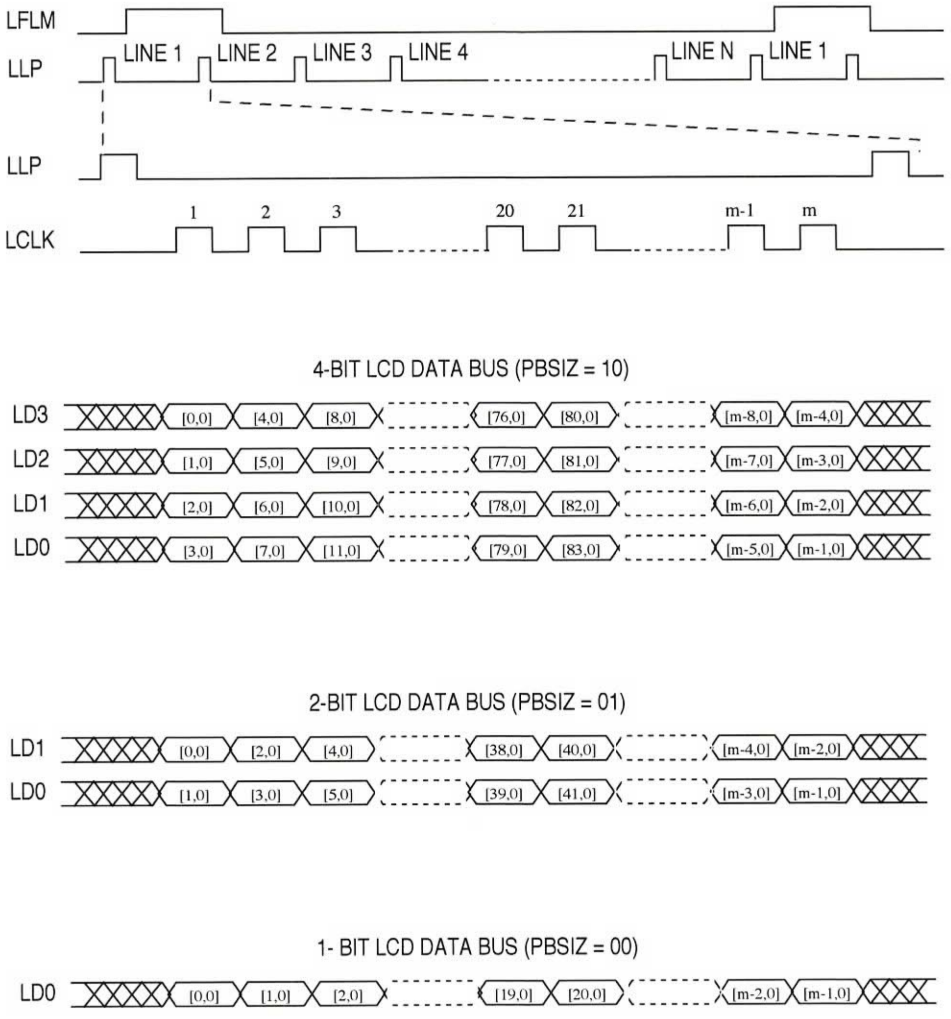

LFLM
LLP
LINE 1
LINE 2
LINE 3
LINE 4
LINE N
LINE 1
LLP
LCLK
1
2
3
20
21
m-1
m
4-BIT LCD DATA BUS (PBSIZ = 10)
LD3 [0,0] [4,0] [8,0] [76,0] [80,0] [m-8,0] [m-4,0]
LD2 [1,0] [5,0] [9,0] [77,0] [81,0] [m-7,0] [m-3,0]
LD1 [2,0] [6,0] [10,0] [78,0] [82,0] [m-6,0] [m-2,0]
LD0 [3,0] [7,0] [11,0] [79,0] [83,0] [m-5,0] [m-1,0]
2-BIT LCD DATA BUS (PBSIZ = 01)
LD1 [0,0] [2,0] [4,0] [38,0] [40,0] [m-4,0] [m-2,0]
LD0 [1,0] [3,0] [5,0] [39,0] [41,0] [m-3,0] [m-1,0]
1- BIT LCD DATA BUS (PBSIZ = 00)
LD0 [0,0] [1,0] [2,0] [19,0] [20,0] [m-2,0] [m-1,0]

Figure 12-2. LCD Interface Timing for 4-, 2-, and 1-Bit Data Widths

12.1.2 Controlling the Display

The LCD controller is designed to drive single-screen monochrome STN LCD panels with up to 640 x 512 pixels in black-and-white display and 320 x 240 in gray level display. Screen size larger than 320 x 240 for gray level display may cause flickering due to slow refresh rate. The best efficiency is achieved when the screen width is a multiple of the DMA controller's 16-bit bus width.

12.1.2.1 FORMAT OF THE LCD SCREEN. The screen width and height of the LCD panel are software programmable. Figure 12-3 illustrates the relationship between the portion of a large graphics file displayed on the screen versus the actual page. The units in the figure are measured in pixel counts.

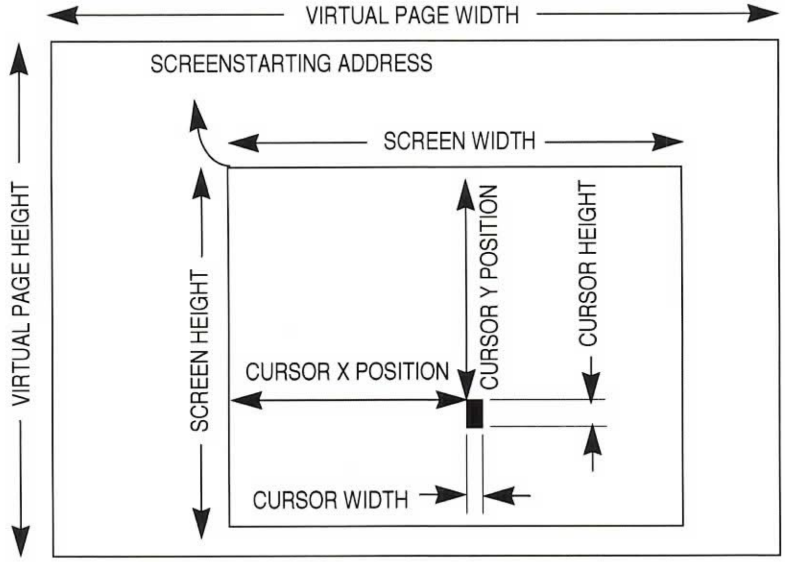


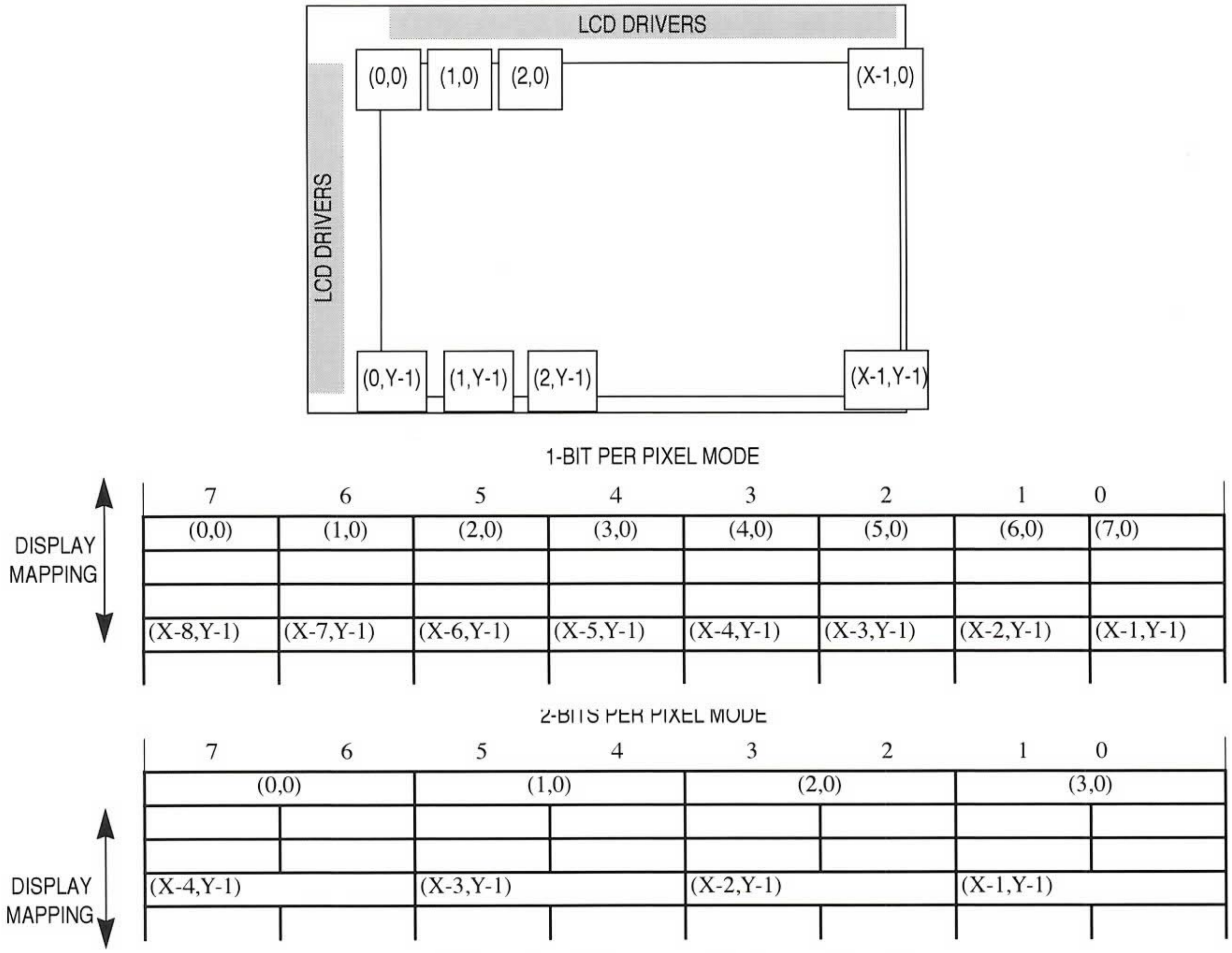
Figure 12-3. LCD Screen Format

The LCD screen width (LXMAX) and LCD screen height (LYMAX) registers are where you specify the size of the LCD panel. The LCD controller will start scanning the display memory at the location pointed to by the LCD screen starting address (LSSA) register. Therefore, the shaded area in Figure 12-3 will be displayed on the LCD panel.

The maximum page width and page height are specified by the LCD virtual page width (LVPW) and LCD virtual page height parameters. By changing the LSSA register, a screen-sized window can be vertically or horizontally scrolled (panned) anywhere inside the virtual page boundaries. However, it is up to your software to position the starting address so that the scanning logic's system memory pointer does not stretch beyond the virtual page width or height. Otherwise, strange objects will appear on the screen. The LVPH parameter shows the bottom of the page, but it is not used by the LCD controller.

12.1.2.2 FORMAT OF THE CURSOR. To define the position of the hardware cursor, the LCD controller maintains a vertical line counter (YCNT) to keep track of the current pixel's vertical position. YCNT, in conjunction with XCNT (the horizontal pixel counter), specifies the screen position of the pixel data being processed. When the pixel falls within a window specified by the cursor's reference position, cursor width, and cursor height, the original pixel bits can be shown with different properties. These properties can be transparent (cursor is disabled), full (black cursor), reversed video, full (white cursor), or blinking. You can make the hardware cursor blink by setting the BKEN bit in the LBLKC register to 1, which alternates the original signal and cursor periodically. You can control the speed at which the cursor blinks by selecting the BDx bit in the LBLKC register. The half-period may be as long as 2 seconds.

12.1.2.3 MAPPING THE DISPLAY DATA. The LCD controller supports 1 or 2 bits per pixel graphics mode. In the 1-bit mode, each bit in the display memory corresponds to a pixel in the LCD panel. The corresponding pixel on the screen is either fully on or fully off. Meanwhile, in 2-bit mode, each pixel is being represented by two bits of display memory. To map the data to the LCD panel, you have to program the appropriate bit in the corresponding address of the display memory. Figure 12-4 illustrates how the system memory data in both modes are mapped.



1-BIT PER PIXEL MODE

7	6	5	4	3	2	1	0
(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(6,0)	(7,0)
(X-8,Y-1)	(X-7,Y-1)	(X-6,Y-1)	(X-5,Y-1)	(X-4,Y-1)	(X-3,Y-1)	(X-2,Y-1)	(X-1,Y-1)

2-BITS PER PIXEL MODE

7	6	5	4	3	2	1	0
(0,0)		(1,0)		(2,0)		(3,0)	
(X-4,Y-1)		(X-3,Y-1)		(X-2,Y-1)		(X-1,Y-1)	

Figure 12-4. Mapping Memory Data on the Screen

12.1.2.4 GENERATING GRAY-SCALE TONES. In 2-bit per pixel mode, a proprietary frame rate control circuitry inside the LCD controller generates intermediate gray-scale tones on the LCD panel by adjusting the density of ones and zeroes that appear over the frames. The LCD controller can generate sixteen simultaneous gray-scale levels out of 16 available palettes. The two levels between black and white can be selected from Table 12-1. Use the LGPMR registers to program the gray-scale level.

Table 12-1. Gray-Scale Palette Options

GRAY-SCALE CODE	DENSITY
0000	0
0001	1/8
0010	3/16
0011	1/4
0100	5/16
0101	3/8
0110	7/16
0111	1/2
1000	9/16
1001	5/8
1010	11/16
1011	3/4
1100	13/16
1101	7/8
1110	15/16
1111	1

NOTE: 0=White and 1=Black.

Since crystal formulations and driving voltages vary, the visual gray-scale effect may or may not be linearly related to the frame rate. For certain types of graphics, a logarithmic scale like 0, $^{1}/_{4}$, $^{1}/_{2}$, and 1, might be more visually pleasing than a linearly spaced scale like 0, $^{5}/_{16}$, $^{11}/_{16}$, and 1. This flexible mapping scheme allows you to optimize the visual effect for the specific panel or application during a sixteen gray-scale level display mode.

12.1.2.5 CONTROLLING FRAME RATE MODULATION. Sometimes blinking or flickering will occur if all of the LCD pixel cells are driven at the same time. To minimize flickering, you can program two 4-bit numbers, specifically the XMOD and YMOD bits in the LCD frame rate modulation control (LFRCM) register. As a general rule, you should select odd numbers and the two values should differ by at least 2. The optimal offset values could vary among LCD panel models (even those by the same manufacturer) because of different inter-pixel cross-talk characteristics. However, the default value of the LFRCM register should work for most of the LCD panels on the market.

12.1.3 Using Low-Power Mode

Some panels may have a PANEL_OFF signal, which is used to turn off the panel for low-power mode. In an MC68EZ328 system, this signal is not supported, but can be easily implemented using a parallel I/O pin. You can program your software to achieve PANEL_OFF by using parallel I/O in the following sequence:

1. Drive the LCD bias voltage to +15V or -15V.
2. Set the LCDON bit to 0 in the LCD clocking control (LCKCON) register, which turns off the LCD controller.

To turn the LCD controller back on, follow these steps:

1. Set the LCDON bit to 1 in the LCKCON register, which turns on the LCD controller.
2. Pause for 1 or 2ms.
3. Drive the LCD bias voltage to +15V or -15V.

When you set the LCDON bit in the CLKCON register to 1, the LCD controller will enter low-power mode by stopping its own pixel clock prior to the next line buffer fill DMA. Further screen DMA and display refresh operations will then be halted in this mode. When the LCD controller is turned back on, DMA and screen refresh activities will resume synchronously.

12.1.4 Using the DMA Controller

This LCD DMA controller is a fly-by type 16-bit wide fast data transfer machine. Since the LCD screen has to be continuously refreshed at a rate of 50-70Hz, the pixel bits in the memory will be read and transferred to the corresponding pixels on the screen. To minimize bus obstruction, a burst type and fly-by transfer is required. Each cycle is evenly distributed across the time frame. Every time the internal line buffer needs data, it asserts the $\overline{\text{BR}}$ signal to request the bus from the core. Once the core grants the bus ($\overline{\text{BG}}$ is asserted), the DMA controller gets control of the bus signal and issues a number of words read from memory. The read data is then internally passed to the internal pixel buffer. During the LCD access cycles, output enable and chip-select signals for the corresponding system memory chip are asserted by the chip-select logic inside the system integration module. You can minimize bus bandwidth obstruction by using zero LCD access wait-states (1 clock per access).

12.1.4.1 BUS BANDWIDTH CALCULATION EXAMPLE. Since LCD screen refresh occurs periodically, the load that the LCD controller puts on the host data bus becomes an important consideration to the high performance handheld system designer. There are many issues involved in estimating bandwidth overhead to the data bus.

Consider a typical scenario:

Screen size: 320 x 240 pixels

Bits per pixel: 2-bits per pixel

Screen refresh rate: 60Hz

System clock: 16.58MHz

Host bus size: 16-bit

DMA access cycle: 2 cycles per 16-bit word

The following T_l period is used by the LCD controller to update one line of the screen:

$$T_l = \frac{1}{60\mathrm{Hz}} \times \frac{1}{240 \text{ lines}}$$
$$= 69.4\mu s$$

During the same period, the line buffer must be filled. The following T_{DMA} duration is how long the DMA cycle will hold up the bus:

$$T_{DMA} = \frac{320 \text{ pixels} \times 2 \text{ bits per pixel} \times 2 \text{ clocks}}{16.67\mathrm{MHz} \times 16\text{-bit bus}}$$
$$= 4.8\mu s$$

Thus, the percentage of host bus time taken up by LCD controller's DMA is P_{DMA} as follows:

$$P_{DMA} = \frac{4.8` \mu s}{69.4` \mu s}$$
$$= 6.92`\%$$

12.2 PROGRAMMING MODEL

12.2.1 LCD Screen Starting Address Register

The LCD screen starting address (LSSA) register is used to inform the LCD panel where to fetch the data to be displayed.

LSSA

BIT	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
FIELD	RESERVED			SSA28	SSA27	SSA26	SSA25	SSA24	SSA23	SSA22	SSA21	SSA20	SSA19	SSA18	SSA17	SSA16
RESET	0x00000000															
ADDR	0x(FF)FFFA00															

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIELD	SSA15	SSA14	SSA13	SSA12	SSA11	SSA10	SSA9	SSA8	SSA7	SSA6	SSA5	SSA4	SSA3	SSA2	SSA1	—
RESET	0x00000000															
ADDR	0x(FF)FFFA00															

Bit 31–29—Reserved

These bits are reserved and must be set to 0.

SSAx—Screen Starting Address 28–1

This field is the 28-bit screen starting address of the LCD panel. The LCD controller will start fetching pixel data from system memory at this address. This field must start at a location that will enable a complete picture to be stored in a 128K memory boundary (A[16:00]). In other words, A[28:17] has a fixed value for a picture's image.

12.2.2 LCD Virtual Page Width Register

The LCD virtual page width (LVPW) register contains the width of the displayed image.

LVPW

BIT	7	6	5	4	3	2	1	0
FIELD	VP8	VP7	VP6	VP5	VP4	VP3	VP2	VP1
RESET	0xFF							
ADDR	0x(FF)FFFA05							

VPx—Virtual Page Width 8–1

This bit specifies the virtual page width of the LCD panel in terms of word count.
The virtual page width is the virtual width in pixels divided by 16 for a black and white display and 8 for a four level gray-scale display and divided by 4 for a sixteen-level grayscale display.

12.2.3 LCD Screen Width Register

The LCD screen width register (LXMAX) is used to define the width of your LCD panel's screen. This register must be a multiple of 16.

LXMAX

BIT	15	14	13	12	11	10	9	8	7	6	55	4	3	2	1	0
FIELD	–						XM9	XM8	XM7	XM6	XM5	XM4	–			
RESET	0x03F0															
ADDR	0x(FF)FFFA08															

XMx—Width Maximum 9–4

These bits represent the width of the LCD panel in number of pixels.

12.2.4 LCD Screen Height Register

The LCD screen height register (LYMAX) is used to define the height of your LCD panel's screen.

LYMAX

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIELD	–							YM8	YM7	YM6	YM5	YM4	YM3	YM2	YM1	YM0
RESET	0x01FF															
ADDR	0x(FF)FFFA0A															

YMx—Height Maximum 8–0

These bits represent the height of the LCD panel in number of pixels, which is equal to YMAX+1.

12.2.5 LCD Cursor X Position Register

The LCD cursor X position (LCXP) register is used to determine the horizontal position of your cursor on the LCD panel.

LCXP

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIELD	CC1	CC0	–				CXP9	CXP8	CXP7	CXP6	CXP5	CXP4	CXP3	CXP2	CXP1	CXP0
RESET	0x0000															
ADDR	0x(FF)FFFA18															

CCx—Cursor Control 1 and 0

These bits are used to control the format of your cursor.

00 = Transparent, cursor is disabled.
01 = Full (black) cursor.
10 = Reversed video.
11 = Full (white) cursor.

CXPx—Cursor X Position 9–0

These bits represent the cursor's horizontal starting position X in pixel count (from 0 to XMAX).

12.2.6 LCD Cursor Y Position Register

The LCD cursor Y position (LCYP) register is used to determine the vertical position of your cursor on the LCD panel.

LCYP

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIELD	–							CYP8	CYP7	CYP6	CYP5	CYP4	CYP3	CYP2	CYP1	CYP0
RESET	0x0000															
ADDR	0x(FF)FFFA1A															

CYPx—Cursor Vertical Y Pixel 8–0

These bits represent the cursor's vertical starting position Y in pixel count (from 0 to YMAX).

12.2.7 LCD Cursor Width and Height Register

The LCD cursor width and height (LCWCH) register is used to determine the width and height of your cursor.

LCWCH

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIELD	–			CW4	CW3	CW2	CW1	CW0	–			CH4	CH3	CH2	CH1	CH0
RESET	0x0101															
ADDR	0x(FF)FFFA1C															

CWx—Cursor Width 4–0

These bits specify the width of the hardware cursor in pixel count (from 1 to 31).

CHx—Cursor Height 4–0

These bits specify the height of the hardware cursor in pixel count (from 1 to 31).

Note: The cursor is disabled if the CWx or CHx bits are set to zero.

12.2.8 LCD Blink Control Register

The LCD blink control register (LBLKC) is used to control how your cursor blinks.

LBLKC

BIT	7	6	5	4	3	2	1	0
FIELD	BKEN	BD6	BD5	BD4	BD3	BD2	BD1	BD0
RESET	0x7F							
ADDR	0x(FF)FFFA1F							

BKEN—Blink Enable

This bit determines if the blink enable cursor will blink or remain steady.

1 = Blink is enabled.
0 = Blink is disabled (default).

BDx—Blink Divisor 6–0

These bits determine if the cursor will toggle once per a specified number of internal frame pulses plus one. The half-period may be as long as 2 seconds.

12.2.9 LCD Panel Interface Configuration Register

The LCD panel interface configuration (LPICF) register is used to determine the data bus width of the LCD panel and to determine if it is a black and white or grayscale display.

LPICF

BIT	7	6	5	4	3	2	1	0
FIELD	–				PBSIZ1	PBSIZ0	GS1	GS0
RESET	0x00							
ADDR	0x(FF)FFFA20							

PBSIZx—Panel Bus Width 1–0

00 = 1-bit.
01 = 2-bit.
10 = 4-bit.
11 = Unused.

GSx—Gray-Scale Mode Selection 1–0

00 = Black and white mode.
01 = Four level gray-scale mode.
10 = Sixteen level gray-scale mode.
11 = Reserved.

12.2.10 LCD Polarity Configuration Register

The LCD polarity configuration (LPOLCF) register controls the polarity of the interface signal that goes to the LCD panel.

LPOLCF

BIT	7	6	5	4	3	2	1	0
FIELD	—				LCKPOL	FLMPOL	LPPOL	PIXPOL
RESET	0x00							
ADDR	0x(FF)FFFA21							

LCKPOL—LCD Shift Clock Polarity

This bit controls the polarity of the active edge of the LCD shift clock.

0 = Active negative edge of LCLK.
1 = Active positive edge of LCLK.

FLMPOL—Frame Marker Polarity

This bit controls the polarity of the frame marker.

0 = Frame marker is active high.
1 = Frame marker is active low.

LPPOL—Line Pulse Polarity

This bit controls the polarity of the line pulse.

0 = Line pulse is active high.
1 = Line pulse is active low.

PIXPOL—Pixel Polarity

This bit controls the polarity of the pixels.

0 = Pixel polarity is active high.
1 = Pixel polarity is active low.

12.2.11 LACD Rate Control Register

The LCD alternate crystal direction rate control (LACDRC) register is used to control the alternate rates of the liquid crystal direction.

LACDRC

BIT	7	6	5	4	3	2	1	0
FIELD	ACDSLT	—			ACD3	ACD2	ACD1	ACD0
RESET	0x00							

LACDRC

BIT	7	6	5	4	3	2	1	0
ADDR	0x(FF)FFFA23							

ACDSLT –Signal Source Select

0 = Select Frame Pulse

1 = Select Line Pulse

ACDx—Alternate Crystal Direction Control 3–0

These bits represent the ACD toggle rate control code. The LACD signal will toggle once every 1 to 16 FLM cycles based on the value specified in this register. The actual number of FLM cycles is the value programmed plus one. Shorter cycles tend to give better results.

12.2.12 LCD Pixel Clock Divider Register

The LCD pixel clock divider (LPXCD) register is used to program the divider, which generates the pixel clock.

LPXCD

BIT	7	6	5	4	3	2	1	0
FIELD	–		PCD5	PCD4	PCD3	PCD2	PCD1	PCD0
RESET	0x00							
ADDR	0x(FF)FFFA25							

PCDx—Pixel Clock Divider 5–0

These bits represent the pixel clock divisor. The LCLK signal from the PLL is divided by N (PCD5-0 plus one) to yield the actual pixel clock. Values of 1-63 will yield N=2 to 64. If set to 0 (N=1), the PIX clock will be used directly, thus bypassing the divider circuit. Refer to for more information.

12.2.13 LCD Clocking Control Register

The LCD clocking control (LCKCON) register is used to enable the LCD controller and control the LCD memory cycle.

LCKCON

BIT	7	6	5	4	3	2	1	0
FIELD	LCDON	DWIDTH	–		DWS3	DWS2	DWS1	DWS0
RESET	0x00							
ADDR	0x(FF)FFFA27							

LCDON—LCD Control

0 = Disable the LCD controller.
1 = Enable the LCD controller.

DWIDTH—Display Memory Width

This bit sets the bus width of the display memory.

0 = 16-bit bus width.
1 = 8-bit bus width.

DWSx—Display Wait-State 3–0

These bits represent the static display memory wait-state control. It is the number of clock cycles per DMA access (for SRAM or ROM only).

0000 = One clock cycle transfer.
0001 = Two clock cycle transfers.
•
•
•
1111 = Sixteen clock cycle transfers.

12.2.14 LCD Refresh Rate Adjustment Register

The LCD refresh rate adjustment (LRRA) register is used to fine-tune the display refresh rate by introducing an idle interval between alternate LCD DMA and display cycles.

LRRA

BIT	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
FIELD	RRA7	RRA6	RRA5	RRA4	RRA3	RRA2	RRA1	RRA0
RESET	0xFF							
ADDR	0x(FF)FFFA29							

RRAx—Refresh Rate 7–0

These bits contain the frame period, which can be calculated as follows:

Frame period = (6+RRA+width) x height x (PCXD+1) x LCLK,

where:

Frame period = Number of nanoseconds for each screen update.
RRAx = Hexadecimal value stored in the LRRA register.
Width = Screen width in number of pixels.
Height = Screen height in number of pixels.
PCXD = Hexadecimal value stored in the LPXCD register.
LCLK = Period in nanoseconds for LCLK.

12.2.15 LCD Panning Offset Register

The LCD panning offset register (LPOSR) is used to control how many pixels the picture is shifted to the left.

LPOSR

BIT	7	6	5	4	3	2	1	0
FIELD	—				POS3	POS2	POS1	POS0
RESET	0x00							
ADDR	0x(FF)FFFA2D							

POSx—Pixel Offset Code

These bits specify the number of pixels being shifted to the left of the display panel.

POS [3:0] - Pixel Offset Code for black-and-white display
POS [2:0] - Pixel Offset Code for four level gray-scale dispaly
POS [1:0] - Pixel Offset Code for sixteen level gray-scale display.

Note: When you modify this register, your software must adjust the cursor's reference position.

12.2.16 LCD Frame Rate Control Modulation Register

The LCD frame rate control modulation register (LFRCM) is used to minimize the flickering on your LCD panel.

LFRCM

BIT	7	6	5	4	3	2	1	0
FIELD	XMOD3	XMOD2	XMOD1	XMOD0	YMOD3	YMOD2	YMOD1	YMOD0
RESET	0xB9							
ADDR	0x(FF)FFFA31							

XMODx—Horizontal Modulation 3–0

These bits modulate adjacent pixels at different time periods to avoid spatial flicker or jitter when frame rate control is used. These values must be optimized by manually fine-tuning the target LCD panel. See Section 12.1.2 Controlling the Display for more information.

YMODx—Vertical Modulation 3–0

These bits modulate adjacent pixels at different time periods to avoid spatial flicker or jitter when frame rate control is used. These values must be optimized by manually fine-tuning the target LCD panel. See Section 12.1.2 Controlling the Display for more information.

12.2.17 LCD Gray Palette Mapping Register

For 4-level gray-scale displays, full black and full white are the two predefined display levels. The other two intermediate gray-scale shading densities can be adjusted here in the LCD gray palette mapping register (LGPMR).

LGPMR

BIT	7	6	5	4	3	2	1	0
FIELD	G23	G22	G21	G20	G13	G12	G11	G10
RESET	0x84							
ADDR	0x(FF)FFFA33							

G23–G20—Gray-Scale 23–20

These bits represent one of the two gray-scale shading densities.

G13–G10—Gray-Scale 13–10

These bits represent the other gray-scale shading density.

12.2.18 PWM Contrast Control Register

The pulse-width modulator contrast control register (PWMR) is used to control PWMO signal, which controls the contrast of the LCD panel.

PWMR

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIELD	–					SCR1	SCR0	CCPE N	PW7	PW6	PW5	PW4	PW3	PW2	PW1	PW0
RESET	0x0000															
ADDR	0x(FF)FFFA36															

SRCx—Source 1–0

These bits select the input clock source for the PWM counter. Therefore, the PWM output frequency is equal to the frequency of the input clock divided by 256.

00 = Line pulse.
01 = Pixel clock.
10 = LCD clock.
11 = Reserved.

CCPEN—Contrast Control Enable

The bit is used to enable or disable the contrast control function.

0 = Contrast control is off.
1 = Contrast control is on.

PWx—Pulse-Width 7–0

This bit controls the pulse-width of the built-in pulse-width modulator, which controls the contrast of your LCD screen. See for more information.

12.3 PROGRAMMING EXAMPLE

The following is an example of how to program the related registers so that you can properly configure an LCD panel with a resolution of 240x160 pixels, four levels of gray-scale, and a 4-bit LCD data interface. The virtual image is 320 pixels wide and panned by three pixels.

```
LCDINT    move.l #$A80000,#$FFFA00 ;display data address starts at $A80000
          move.w #240,#$FFFA08     ;LCD horizontal size is 240
          move.w #159,#$FFFA0A     ;LCD vertical size is 160
          move.b #40,#$FFFA05      ;4 level grey and 320 pixels wide image
          move.b #$09,#$FFFA20     ;LCD panel data bus is 4 bits, 4 level grey
          move.b #3,#$FFFA25       ;pixel clock rate equal 1/4 of LCDCLK from PLL
          move.b #10,#$FFFA29      ;refresh rate adjustment
          move.b #$03,#$FFFA2D     ;shift picture by 3 pixels
          move.b #$82,#$FFFA27     ;switch on LCDC, 2 wait state for memory cycle
```

Pilot PDA

Re: Pilot: IIIc CPU questions repost

Search: Search
In:dents Matching:All words

Home Page
Year Index

Thread Index
Date Index
Author Index

Date Previous
Date Next
Thread Previous
Thread Next

From Chris Terwilliger <chris@searat.com>
To pilot@ultraviolet.org
Subject Re: Pilot: IIIc CPU questions repost
Date Mon, 28 Feb 2000 22:26:34 -0500

Chris Mauricio wrote:

> Hey all- still looking for info
>
> So reading the specs on the IIIc, it shows a DragonBall 20 Mhz CPU, I
> understood it was rumoured to use the 33 Mhz CPU.. any comments? Any idea

The IIIc is using the 68EZ328 which is spec'd at 20Mhz. This is the same cpu/speed as the Vx. The 33Mhz part is the new 68VZ328 which no current Palm device is using. Why is that you ask? Doesn't the VZ part support color? Why yes it does, but it only supports lower quality STN type color displays and only 16 colors. So Palm teamed the older EZ part with the Epson SED1375 Embedded Memory LCD Controller. The SED1375 is a really cool display driver with a lot of features and is designed to be directly driven by a dragonball. It supports active matrix TFT panels and up to 256 colors. It has 80K of embedded RAM so you don't have to use system memory for your display buffer. It also does display rotation and virtual display (panning & scrolling) on chip. Just what Palm needed for the IIIc. Of course later Palm models *could* swap the EZ part for the VZ part at 33Mhz and get a fairly dramatic speedup and still use the SED1375 for the display.

--
// Chris Terwilliger
// chris@searat.com
// zoom@palm.net

The Pilot list/archive/unsubscribe page is http://www.ultraviolet.org

---

- **References**:
  - **Pilot: IIIc CPU questions repost**
    - *From:* "Chris Mauricio" <cmauricio@arrk.com>

- Prev by Date: **Re: Pilot: Palm Store -- One Word: Cluster*****
- Next by Date: **Re: Pilot: Outlook Express 5 and Plam Mail**
- Prev by thread: **Pilot: IIIc CPU questions repost**
- Next by thread: **Pilot: Profane Language**
- Index(es):
  - **Date**

8 Registers

8.1 Register Mapping

The SED1375 registers are located in the upper 32 bytes of the 128K byte SED1375 address range. The registers are accessible when CS# = 0 and AB[16:0] are in the range 1FFE0h through 1FFFFh.

Base = 0x1F00 0000
(IIIc)

8.2 Register Descriptions

Unless specified otherwise, all register bits are reset to 0 during power up.
All bits marked n/a should be programmed 0.

REG[00h] Revision Code Register
Address = 1FFE0h Read Only.

Product Code Bit 5	Product Code Bit 4	Product Code Bit 3	Product Code Bit 2	Product Code Bit 1	Product Code Bit 0	Revision Code Bit 1	Revision Code Bit 0

bits 7-2 Product Code
This is a read-only register that indicates the product code of the chip. The product code is 001001.

bits 1-0 Revision Code
This is a read-only register that indicates the revision code of the chip. The revision code is 00.

REG[01h] Mode Register 0
Address = 1FFE1h Read/Write.

TFT/STN	Dual/Single	Color/Mono	FPLine Polarity	FPFrame Polarity	Mask FPSHIFT	Data Width Bit 1	Data Width Bit 0

bit 7 TFT/STN
When this bit = 0, STN (passive) panel mode is selected. When this bit = 1, TFT/D-TFD panel mode is selected. If TFT/D-TFD panel mode is selected, Dual/Single (REG[01h] bit 6) and Color/Mono (REG[01h] bit5) are ignored. See Table 8-1: "Panel Data Format" for a comprehensive description of panel selection.

bit 6 Dual/Single
When this bit = 0, Single LCD panel drive is selected. When this bit = 1, Dual LCD panel drive is selected. See Table 8-1: "Panel Data Format" for a comprehensive description of panel selection.

bit 5 Color/Mono
When this bit = 0, Monochrome LCD panel drive is selected. When this bit = 1, Color LCD panel drive is selected. See Table 8-1: "Panel Data Format" for a comprehensive description of panel selection.

bit 4 FPLINE Polarity
This bit controls the polarity of FPLINE in TFT/D-TFD mode (no effect in passive panel mode). When this bit = 0, FPLINE is active low. When this bit = 1, FPLINE is active high.

bit 3 FPFRAME Polarity
This bit controls the polarity of FPFRAME in TFT/D-TFD mode (no effect in passive panel mode). When this bit = 0, FPFRAME is active low. When this bit = 1, FPFRAME is active high.

bit 2 Mask FPSHIFT
FPSHIFT is masked during non-display periods if either of the following two criteria is met:

1. Color passive panel is selected (REG[01h] bit 5 = 1)
2. This bit (REG[01h] bit 2) = 1

bits 1-0 Data Width Bits [1:0]
These bits select the display data format. See Table 8-1: "Panel Data Format" below for a comprehensive description of panel selection.

Table 8-1: Panel Data Format

TFT/STN	**Color/Mono**	**Dual/Single**	**Data Width Bit 1**	**Data Width Bit 0**	**Function**
REG[01h] bit 7	REG[01h] bit 5	REG[01h] bit 6	REG[01h] bit 1	REG[01h] bit 0	
0	0	0	0	0	Mono Single 4-bit passive LCD
				1	Mono Single 8-bit passive LCD
			1	0	reserved
				1	reserved
		1	0	0	reserved
				1	Mono Dual 8-bit passive LCD
			1	0	reserved
				1	reserved
	1	0	0	0	Color Single 4-bit passive LCD
				1	Color Single 8-bit passive LCD format 1
			1	0	reserved
				1	Color Single 8-bit passive LCD format 2
		1	0	0	reserved
				1	Color Dual 8-bit passive LCD
			1	0	reserved
				1	reserved
1	X (don't care)			0	9-bit TFT/D-TFD panel
				1	12-bit TFT/D-TFD panel

REG[02h] Mode Register 1
Address = 1FFE2h Read/Write.

Bit-Per-Pixel Bit 1	Bit-Per-Pixel Bit 0	High Performance	Input Clock divide (CLKI/2)	Display Blank	Frame Repeat	Hardware Video Invert Enable	Software Video Invert

bits 7-6 Bit-Per-Pixel Bits [1:0]
These bits select the color or gray-scale depth (Display Mode).

Table 8-2: Gray Scale/Color Mode Selection

Color/Mono **REG[01h] bit 5**	**Bit-Per-Pixel Bit 1** **REG[02h] bit 7**	**Bit-Per-Pixel Bit 0** **REG[02h] bit 6**	**Display Mode**	
0	0	0	2 Gray scale	1 bit-per-pixel
		1	4 Gray scale	2 bit-per-pixel
	1	0	16 Gray scale	4 bit-per-pixel
		1	reserved	
1	0	0	2 Colors	1 bit-per-pixel
		1	4 Colors	2 bit-per-pixel
	1	0	16 Colors	4 bit-per-pixel
		1	256 Colors	8 bit-per-pixel

bit 5 High Performance (Landscape Modes Only)
When this bit = 0, the internal Memory Clock (MCLK) is a divided-down version of the Pixel Clock (PCLK). The denominator is dependent on the bit-per-pixel mode - see the table below.

Table 8-3: High Performance Selection

High Performance	**BPP Bit 1**	**BPP Bit 0**	**Display Modes**	
0	0	0	MClk = PClk/8	1 bit-per-pixel
		1	MClk = PClk/4	2 bit-per-pixel
	1	0	MClk = PClk/2	4 bit-per-pixel
		1	MClk = PClk	8 bit-per-pixel
1	X	X	MClk = PClk	

When this bit = 1, MCLK is fixed to the same frequency as PCLK for all bit-per-pixel modes. This provides a faster screen update performance in 1/2/4 bit-per-pixel modes, but also increases power consumption. This bit can be set to 1 just before a major screen update, then set back to 0 to save power after the update. This bit has no effect in SwivelView mode. Refer to REG[1Bh] SwivelView Mode Register on page 66 for SwivelView mode clock selection.

bit 4

Input Clock Divide
When this bit = 0, the Operating Clock(CLK) is the same as the Input Clock (CLKI). When this bit = 1, CLK = CLKI/2.

In landscape mode PCLK=CLK and MCLK is selected as per Table 8-3: "High Performance Selection".

In SwivelView mode, MCLK and PCLK are derived from CLK as shown in Table 8-8: "Selection of PCLK and MCLK in SwivelView Mode," on page 67.

bit 3

Display Blank
This bit blanks the display image. When this bit = 1, the display is blanked (FPDAT lines to the panel are driven low). When this bit = 0, the display is enabled.

bit 2

Frame Repeat (EL support)
This feature is used to improve Frame Rate Modulation of EL panels. When this bit = 1, an internal frame counter runs from 0 to 3FFFFh. When the frame counter rolls over, the modulated image pattern is repeated (every 1 hour when the frame rate is 72Hz). When this bit = 0, the modulated image pattern is never repeated.

bit 1

Hardware Video Invert Enable
In passive panel modes (REG[01h] bit 7 = 0) FPDAT11 is available as either GPIO4 or hardware video invert. When this bit = 1, Hardware Video Invert is enabled via the FPDAT11 pin. When this bit = 0, FPDAT11 operates as GPIO4. See Table 8-4: "Inverse Video Mode Select Options" below.

Note
Video data is inverted after the Look-Up Table.

bit 0

Software Video Invert
When this bit = 1, Inverse Video Mode is selected. When this bit = 0, Standard Video Mode is selected. See Table 8-4: "Inverse Video Mode Select Options" below.

Note
Video data is inverted after the Look-Up Table.

Table 8-4: Inverse Video Mode Select Options

Hardware Video Invert Enable	Software Video Invert (Passive and Active Panels)	FPDAT11 (Passive Panels Only)	Video Data
0	0	X	Normal
0	1	X	Inverse
1	X	0	Normal
1	X	1	Inverse

REG[03h] Mode Register 2 Address = 1FFE3h							Read/Write
n/a	n/a	n/a	n/a	LCDPWR Override	Hardware Power Save Enable	Software Power Save Bit 1	Software Power Save Bit 0

bit 3 LCDPWR Override
This bit is used to override the panel on/off sequencing logic. When this bit = 0, LCDPWR and the panel interface signals are controlled by the sequencing logic. When this bit 1, LCDPWR is forced to off and the panel interface signals are forced low immediately upon entering power save mode. See Section 7.3.2, "Power Down/Up Timing" on page 36 for further information.

bit 2 Hardware Power Save Enable
When this bit = 1 GPIO0 is used as the Hardware Power Save input pin. When this bit = 0, GPIO0 operates normally.

Table 8-5: Hardware Power Save/GPIO0 Operation

RESET# State	**Hardware Power Save Enable** **REG[03h] bit 2**	**GPIO0 Config** **REG[18h] bit 0**	**GPIO0 Status/Control** **REG[19h] bit 0**	**GPIO0 Operation**
0	X	X	X	
1	0	0	reads pin status	GPIO0 Input (high impedance)
1	0	1	0	GPIO0 Output = 0
1	0	1	1	GPIO0 Output = 1
1	1	X	X	Hardware Power Save Input (active high)

bits 1-0 Software Power Save Bits [1: 0]
These bits select the Power Save Mode as shown in the following table.

Table 8-6: Software Power Save Mode Selection

Bit 1	**Bit 0**	**Mode**
0	0	Software Power Save
0	1	reserved
1	0	reserved
1	1	Normal Operation

Refer to Section 13, "Power Save Modes" on page 81 for a complete description of the power save modes.

REG[04h] Horizontal Panel Size Register Address = 1FFE4h							Read/Write
n/a	Horizontal Panel Size Bit 6	Horizontal Panel Size Bit 5	Horizontal Panel Size Bit 4	Horizontal Panel Size Bit 3	Horizontal Panel Size Bit 2	Horizontal Panel Size Bit 1	Horizontal Panel Size Bit 0

bits 6-0 Horizontal Panel Size Bits [6:0]

This register determines the horizontal resolution of the panel. This register must be programmed with a value calculated as follows:

$$\text{HorizontalPanelSizeRegister} = \left(\frac{\text{HorizontalPanelResolution(pixels)}}{8}\right) - 1$$

Note

This register must not be set to a value less than 03h.

REG[05h] Vertical Panel Size Register (LSB) Address = 1FFE5h							Read/Write
Vertical Panel Size Bit 7	Vertical Panel Size Bit 6	Vertical Panel Size Bit 5	Vertical Panel Size Bit 4	Vertical Panel Size Bit 3	Vertical Panel Size Bit 2	Vertical Panel Size Bit 1	Vertical Panel Size Bit 0

REG[06h] Vertical Panel Size Register (MSB) Address = 1FFE6h							Read/Write
n/a	n/a	n/a	n/a	n/a	n/a	Vertical Panel Size Bit 9	Vertical Panel Size Bit 8

REG[05h] bits 7-0 Vertical Panel Size Bits [9:0]

REG[06h] bits 1-0 This 10-bit register determines the vertical resolution of the panel. This register must be programmed with a value calculated as follows:

$$\text{VerticalPanelSizeRegister} = \text{VerticalPanelResolution(lines)} - 1$$

3FFh is the maximum value of this register for a vertical resolution of 1024 lines.

REG[07h] FPLINE Start Position Address = 1FFE7h							Read/Write
n/a	n/a	n/a	FPLINE Start Position Bit 4	FPLINE Start Position Bit 3	FPLINE Start Position Bit 2	FPLINE Start Position Bit 1	FPLINE Start Position Bit 0

bits 4-0 FPLINE Start Position
These bits are used in TFT/D-TFD mode to specify the position of the FPLINE pulse. These bits specify the delay, in 8-pixel resolution, from the end of a line of display data (FPDAT) to the leading edge of FPLINE. This register is effective in TFT/D-TFD mode only (REG[01h] bit 7 = 1). This register is programmed as follows:

$$\text{FPLINEposition(pixels)} = (\text{REG[07h]} + 2) \times 8$$

The following constraint must be satisfied:

$$\text{REG[07h]} \le \text{REG[08h]}$$

REG[08h] Horizontal Non-Display Period Address = 1FFE8h							Read/Write
n/a	n/a	n/a	Horizontal Non-Display Period Bit 4	Horizontal Non-Display Period Bit 3	Horizontal Non-Display Period Bit 2	Horizontal Non-Display Period Bit 1	Horizontal Non-Display Period Bit 0

bits 4-0 Horizontal Non-Display Period
These bits specify the horizontal non-display period in 8-pixel resolution.

$$\text{HorizontalNonDisplayPeriod(pixels)} = (\text{REG[08h]} + 4) \times 8$$

REG[09h] FPFRAME Start Position Address = 1FFE9h							Read/Write
n/a	n/a	FPFRAME Start Position Bit 5	FPFRAME Start Position Bit 4	FPFRAME Start Position Bit 3	FPFRAME Start Position Bit 2	FPFRAME Start Position Bit 1	FPFRAME Start Position Bit 0

bits 5-0 FPFRAME Start Position
These bits are used in TFT/D-TFD mode to specify the position of the FPFRAME pulse. These bits specify the number of lines between the last line of display data (FPDAT) and the leading edge of FPFRAME. This register is effective in TFT/D-TFD mode only (REG[01h] bit 7 = 1). This register is programmed as follows:

$$\text{FPFRAMEposition(lines)= REG[09h]}$$

The contents of this register must be greater than zero and less than or equal to the Vertical Non-Display Period Register, i.e.

$$1 \le \text{REG[09h]} \le \text{REG[0Ah]}$$

REG[0Ah] Vertical Non-Display Period Address = 1FFEAh							Read/Write
Vertical Non-Display Status	n/a	Vertical Non-Display Period Bit 5	Vertical Non-Display Period Bit 4	Vertical Non-Display Period Bit 3	Vertical Non-Display Period Bit 2	Vertical Non-Display Period Bit 1	Vertical Non-Display Period Bit 0

bit 7 Vertical Non-Display Status
This bit =1 during the Vertical Non-Display period.

bits 5-0 Vertical Non-Display Period
These bits specify the vertical non-display period. This register is programmed as follows:

$$\text{VerticalNonDisplayPeriod(lines)} = \text{REG[0Ah] bits [5:0]}$$

Note
This register should be set only once, on power-up during initialization.

REG[0Bh] MOD Rate Register Address = 1FFEBh							Read/Write
n/a	n/a	MOD Rate Bit 5	MOD Rate Bit 4	MOD Rate Bit 3	MOD Rate Bit 2	MOD Rate Bit 1	MOD Rate Bit 0

bits 5-0 MOD Rate Bits [5:0]
When the value of this register is 0, the MOD output signal toggles every FPFRAME. For a non-zero value, the value in this register + 1 specifies the number of FPLINEs between toggles of the MOD output signal. These bits are for passive LCD panels only.

REG[0Ch] Screen 1 Start Address Register (LSB)
Address = 1FFECh Read/Write

Screen 1 Start Address Bit 7	Screen 1 Start Address Bit 6	Screen 1 Start Address Bit 5	Screen 1 Start Address Bit 4	Screen 1 Start Address Bit 3	Screen 1 Start Address Bit 2	Screen 1 Start Address Bit 1	Screen 1 Start Address Bit 0

REG[0Dh] Screen 1 Start Address Register (MSB)
Address = 1FFEDh Read/Write

Screen 1 Start Address Bit 15	Screen 1 Start Address Bit 14	Screen 1 Start Address Bit 13	Screen 1 Start Address Bit 12	Screen 1 Start Address Bit 11	Screen 1 Start Address Bit 10	Screen 1 Start Address Bit 9	Screen 1 Start Address Bit 8

REG[0Dh] bits 7-0
REG[0Ch] bits 7-0

Screen 1 Start Address Bits [15:0]
These bits determine the **word address** of the start of Screen 1 in Landscape modes or the **byte address** of the start of Screen 1 in SwivelView modes.

bytes/2

Note
For SwivelView mode the most significant bit (bit 16) is located in REG[10h].

REG[0Eh] Screen 2 Start Address Register (LSB)
Address = 1FFEEh Read/Write

Screen 2 Start Address Bit 7	Screen 2 Start Address Bit 6	Screen 2 Start Address Bit 5	Screen 2 Start Address Bit 4	Screen 2 Start Address Bit 3	Screen 2 Start Address Bit 2	Screen 2 Start Address Bit 1	Screen 2 Start Address Bit 0

REG[0Fh] Screen 2 Start Address Register (MSB)
Address = 1FFEFh Read/Write

Screen 2 Start Address Bit 15	Screen 2 Start Address Bit 14	Screen 2 Start Address Bit 13	Screen 2 Start Address Bit 12	Screen 2 Start Address Bit 11	Screen 2 Start Address Bit 10	Screen 2 Start Address Bit 9	Screen 2 Start Address Bit 8

REG[0Fh] bits 7-0
REG[0Eh] bits 7-0

Screen 2 Start Address Bits [15:0]
These bits determine the **word address** of the start of Screen 2 in Landscape modes only and has no effect in SwivelView modes.

REG[10h] Screen Start Address Overflow Register
Address = 1FFF0h Read/Write

n/a	n/a	n/a	n/a	n/a	n/a	n/a	Screen 1 Start Address Bit 16

bit 0

Screen 1 Start Address Bit 16
This bit is the most significant bit of Screen 1 Start Address for SwivelView mode. This bit has no effect in Landscape mode.

REG[11h] Memory Address Offset Register Address = 1FFF1h							Read/Write
Memory Address Offset Bit 7	Memory Address Offset Bit 6	Memory Address Offset Bit 5	Memory Address Offset Bit 4	Memory Address Offset Bit 3	Memory Address Offset Bit 2	Memory Address Offset Bit 1	Memory Address Offset Bit 0

bits 7-0

Memory Address Offset Bits [7:0] (Landscape Modes Only)
This register is used to create a virtual image by setting a word offset between the last address of one line and the first address of the following line. If this register is not equal to zero, then a virtual image is formed. The displayed image is a window into the larger virtual image. See Figure 8-1: “Screen-Register Relationship, Split Screen,” on page 64.

This register has no effect in SwivelView modes. See “REG[1Ch] Line Byte Count Register for SwivelView Mode” on page 67.

REG[12h] Screen 1 Vertical Size Register (LSB) Address = 1FFF2h							Read/Write
Screen 1 Vertical Size Bit 7	Screen 1 Vertical Size Bit 6	Screen 1 Vertical Size Bit 5	Screen 1 Vertical Size Bit 4	Screen 1 Vertical Size Bit 3	Screen 1 Vertical Size Bit 2	Screen 1 Vertical Size Bit 1	Screen 1 Vertical Size Bit 0

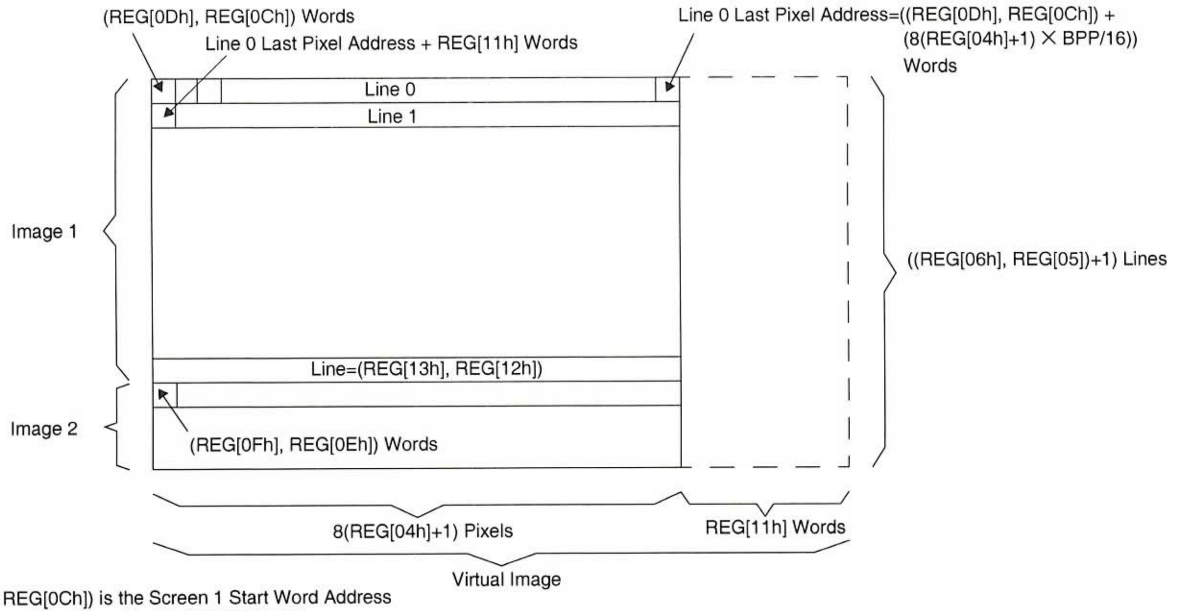
REG[13h] Screen 1 Vertical Size Register (MSB) Address = 1FFF3h							Read/Write
n/a	n/a	n/a	n/a	n/a	n/a	Screen 1 Vertical Size Bit 9	Screen 1 Vertical Size Bit 8

REG[13h] bits 1-0
REG[12h] bits 7-0

Screen 1 Vertical Size Bits [9:0]
This register is used to implement the Split Screen feature of the SED1375. These bits determine the height (in lines) of Screen 1.

In landscape modes, if this register is programmed with a value, n, where n is less than the Vertical Panel Size (REG[06h], REG[05h]), then lines 0 to n of the panel contain Screen 1 and lines n+1 to REG[06h], REG[05h] of the panel contain Screen 2. See Figure 8-1: “Screen-Register Relationship, Split Screen,” on page 64. If Split Screen is not desired, this register must be programmed greater than, or equal to the Vertical Panel Size, REG[06h] and REG[05h].

In SwivelView modes this register must be programmed greater than, or equal to the Vertical Panel Size, REG[06h] and REG[05h]. See “SwivelView™” on page 76.


(REG[0Dh], REG[0Ch]) Words
Line 0 Last Pixel Address + REG[11h] Words
Line 0 Last Pixel Address=((REG[0Dh], REG[0Ch]) + (8(REG[04h]+1) × BPP/16)) Words
Line 0
Line 1
Image 1
((REG[06h], REG[05])+1) Lines
Line=(REG[13h], REG[12h])
Image 2
(REG[0Fh], REG[0Eh]) Words
8(REG[04h]+1) Pixels
REG[11h] Words
Virtual Image

Where:
(REG[0Dh], REG[0Ch]) is the Screen 1 Start Word Address
BPP is Bits-per-Pixel as set by REG[02h] bits 7:6
REG[11h] is the Address Pitch Adjustment in Words
(REG[0Fh], REG[0Eh]) is the Screen 2 Start Word Address
(REG[13h], REG[12h]) is the Screen 1 Vertical Size
(REG[06h], REG[05h]) is the Vertical Panel Size

Figure 8-1: Screen-Register Relationship, Split Screen

Consider an example where REG[13h], REG[12] = 0CEh for a 320x240 display system. The upper 207 lines (CEh + 1) of the panel show an image from the Screen 1 Start Word Address. The remaining 33 lines show an image from the Screen 2 Start Word Address.

REG[15h] Look-Up Table Address Register
Address = 1FFF5h — Read/Write

LUT Address Bit 7	LUT Address Bit 6	LUT Address Bit 5	LUT Address Bit 4	LUT Address Bit 3	LUT Address Bit 2	LUT Address Bit 1	LUT Address Bit 0

bits 7-0 LUT Address Bits [7:0]
These 8 bits control a pointer into the Look-Up Tables (LUT). The SED1375 has three 256-position, 4-bit wide LUTs, one for each of red, green, and blue – refer to Section 11, "Look-Up Table Architecture" on page 70 for details.

This register selects which LUT entry is read/write accessible through the LUT Data Register (REG[17h]). Writing the LUT Address Register automatically sets the pointer to the Red LUT. Accesses to the LUT Data Register automatically increment the pointer.

For example, writing a value 03h into the LUT Address Register sets the pointer to R[3]. A subsequent access to the LUT Data Register accesses R[3] and moves the pointer onto G[3]. Subsequent accesses to the LUT Data Register move the pointer onto B[3], R[4], G[4], B[4], R[5], etc.

Note

The RGB data is inserted into the LUT after the Blue data is written, i.e. all three colors must be written before the LUT is updated.

REG[17h] Look-Up Table Data Register Address = 1FFF7h							Read/Write
LUT Data Bit 3	LUT Data Bit 2	LUT Data Bit 1	LUT Data Bit 0	n/a	n/a	n/a	n/a

bits 7-4 LUT Data Bits [3:0]

This register is used to read/write the RGB Look-Up Tables. This register accesses the entry at the pointer controlled by the Look-Up Table Address Register (REG[15h]).

Accesses to the Look-Up Table Data Register automatically increment the pointer.

Note

The RGB data is inserted into the LUT after the Blue data is written, i.e. all three colors must be written before the LUT is updated.

REG[18h] GPIO Configuration Control Register Address = 1FFF8h							Read/Write
n/a	n/a	n/a	GPIO4 Pin IO Configuration	GPIO3 Pin IO Configuration	GPIO2 Pin IO Configuration	GPIO1 Pin IO Configuration	GPIO0 Pin IO Configuration

bits 4-0 GPIO[4:0] Pin IO Configuration

These bits determine the direction of the GPIO[4:0] pins.

When the GPIOn Pin IO Configuration bit = 0, the corresponding GPIOn pin is configured as an input. The input can be read at the GPIOn Status/Control Register bit. See REG[19h] GPIO Status/Control Register.

When the GPIOn Pin IO Configuration bit = 1, the corresponding GPIOn pin is configured as an output. The output can be controlled by writing the GPIOn Status/Control Register bit.

Note

These bits have no effect when the GPIOn pin is configured for a specific function (i.e. as FPDAT[11:8] for TFT/D-TFD operation).

When configured as IO, all unused pins must be tied to IO V_{DD}.

REG[19h] GPIO Status/Control Register Address = 1FFF9h							Read/Write
n/a	n/a	n/a	GPIO4 Pin IO Status	GPIO3 Pin IO Status	GPIO2 Pin IO Status	GPIO1 Pin IO Status	GPIO0 Pin IO Status

bits 4-0

GPIO[4:0] Status
When the GPIOn pin is configured as an input, the corresponding GPIO Status bit is used to read the pin input. See REG[18h] above.

When the GPIOn pin is configured as an output, the corresponding GPIO Status bit is used to control the pin output.

REG[1Ah] Scratch Pad Register Address = 1FFFAh							Read/Write
Scratch bit 7	Scratch bit 6	Scratch bit 5	Scratch bit 4	Scratch bit 3	Scratch bit 2	Scratch bit 1	Scratch bit 0

bits 7-0

Scratch Pad Register
This register contains general use read/write bits. These bits have no effect on hardware.

REG[1Bh] SwivelView Mode Register Address = 1FFFBh							Read/Write
SwivelView Mode Enable	SwivelView Mode Select	n/a	n/a	n/a	reserved	SwivelView Mode Pixel Clock Select Bit 1	SwivelView Mode Pixel Clock Select Bit 0

bit 7

SwivelView Mode Enable
When this bit = 1, SwivelView Mode is enabled. When this bit = 0, Landscape Mode is enabled.

bit 6

SwivelView Mode Select
When this bit = 0, Default SwivelView Mode is selected. When this bit = 1, Alternate SwivelView Mode is selected. See Section 12, "SwivelView™" on page 76 for further information on SwivelView Mode.

The following table shows the selection of SwivelView Mode.

Table 8-7: Selection of SwivelView Mode

SwivelView Mode Enable (REG[1Bh] bit 7)	**SwivelView Mode Select** (REG[1Bh] bit 6)	**Mode**
0	X	Landscape
1	0	Default SwivelView
1	1	Alternate SwivelView

bit 2
reserved
reserved bits must be set to 0.

bits 1-0
SwivelView Mode Pixel Clock Select Bits [1:0]
These two bits select the Pixel Clock (PCLK) source in SwivelView Mode - these bits have no effect in Landscape Mode. The following table shows the selection of PCLK and MCLK in SwivelView Mode - see Section 12, "SwivelView™" on page 76 for details.

Table 8-8: Selection of PCLK and MCLK in SwivelView Mode

SwivelView Mode Enable (REG[1Bh] bit 7)	**SwivelView Mode Select** (REG[1Bh] bit 6)	**Pixel Clock (PCLK) Select** (REG[1Bh] bits [1:0])		**PCLK =**	**MCLK =**
		Bit 1	**Bit 0**		
0	X	X	X	CLK	See Reg[02h] bit 5
1	0	0	0	CLK	CLK
1	0	0	1	CLK/2	CLK/2
1	0	1	0	CLK/4	CLK/4
1	0	1	1	CLK/8	CLK/8
1	1	0	0	CLK/2	CLK
1	1	0	1	CLK/2	CLK
1	1	1	0	CLK/4	CLK/2
1	1	1	1	CLK/8	CLK/4

Where CLK is CLKI (REG[02h] bit 4 = 0) or CLKI/2 (REG[02h] bit 4 = 1)

REG[1Ch] Line Byte Count Register for SwivelView Mode
Address = 1FFFCh Read/Write

Line Byte Count bit 7	Line Byte Count bit 6	Line Byte Count bit 5	Line Byte Count bit 4	Line Byte Count bit 3	Line Byte Count bit 2	Line Byte Count bit 1	Line Byte Count bit 0

bits 7-0
Line Byte Count Bits [7:0]
This register is the byte count from the beginning of one line to the beginning of the next consecutive line (commonly called "stride" by programmers). This register may be used to create a virtual image in SwivelView mode.

When this register = 00 the "stride" = 256 bytes. This value is used for 240x320 8 bpp default SwivelView mode

When the Line Byte Count Register = n, where 1 ≤ n ≤ FFh, the "stride" = n bytes.

REG[1Eh] and REG[1Fh]

REG[1Eh] and REG[1Fh] are reserved for factory SED1375 testing and should not be written. Any value written to these registers may result in damage to the SED1375 and/or any panel connected to the SED1375.