PPSM

Personal Portable System Manager

Motorola Semiconductor Products Sector

MOTOROLA

Document Number	PPSM Version	Release Date	Comment
PDAPSM01U18-10	2.0	November 15, 1995	
PDAPSM01U18-11	2.1	May 13, 1996	Addendum to PPSM V2.0
PDAPSM03SPM1-10	3.0	March 5, 1997	
PDAPSM03SPM1-11	**3.1**	**November 15, 1998**	

Produced by DragonBall Operation, WSSG, Motorola

Fax: (852) 2666-6551

Email: portable@email.sps.mot.com

Website: http://www.apspg.com/products/ppsm/ppsm.html

Table of Contents

Preface

What is PPSM?

Personal Portable System Manager (PPSM) is a compact operating system designed specifically for the DragonBall™ family of the micro-controllers. This operating system enables most handheld electronic products with LCD displays such as advanced pagers, advanced cellular phones, game machines, GPS, instruments, organizers, and Personal Digital Assistants (PDA).

Audience & Purpose

The first three parts of this manual are oriented to those software engineers wanting to learn the important "rules" of PPSM programming. Each chapter concentrates on a specific aspect of the PPSM Application Programming Interface (API).

The forth part of this manual is oriented to system integrators who are responsible for building and configuring PPSM systems.

Part I

"PPSM Architecture" describes the PPSM environment and architecture.

Part II

"Writing PPSM Applications" concentrates on the features supported by PPSM for application development. Examples are given to better illustrate the methodology. This part is targeted particularly to beginners who are exploring the use of PPSM tools. Therefore, the tools in each chapter are arranged in the suggested order of tools for writing PPSM applications.

Part III

"API Toolset" details the usage of each of the tools. This part is targeted particularly to those experienced engineers who are familiar with the PPSM programming methodology. Therefore, the tools in each chapters are arranged in alphabetical order for quick referencing.

Part IV

"System Integrator□ Guide" focuses on assisting the system integrators to build and configure PPSM system by providing the necessary background information.

Appendices

At the end of the manual, "Appendices" is included to supplement the manual with information.

Part I PPSM Architecture

Chapter 1 Introduction

1.1 What is PPSM?

Personal Portable System Manager (PPSM) is a compact operating system designed specifically for the DragonBall^{TM} family of the microcontrollers. This operating system enables most handheld electronic products with LCD displays such as advanced pagers, advanced cellular phones, game machines, GPS, instruments, organizers, and Personal Digital Assistants (PDA).

PPSM is a real time 32-bit kernel with prioritized interrupt scheduling. All tasks are interrupt-driven, e.g. applications are activated by selecting the corresponding icons. Because PPSM is written in C, it’s highly portable. It contains thorough, configurable, and easy to learn toolsets which aid developers in their application development process.

The PPSM Tools consist of Pen Input, Graphics, Database, Text, Character Input, System and Communication. Application developers have the freedom to design a sophisticated User-Interface and to configure DragonBall^{TM} with easy-to-use API for LCD based products. The PPSM Toolset, together with its Device Drivers, provides the basic control of the LCD, the drawing functions, the real time clock and the UART.

The PPSM kernel does not access hardware devices directly. All peripheral devices are controlled by the kernel indirectly through software device drivers. By supplying the appropriate device drivers with each peripheral, it gives system integrators greater flexibility to use various types of hardware devices without changing the core of the software. *Figure 1-1* shows the architecture of PPSM on the DragonBall^{TM} family platform.

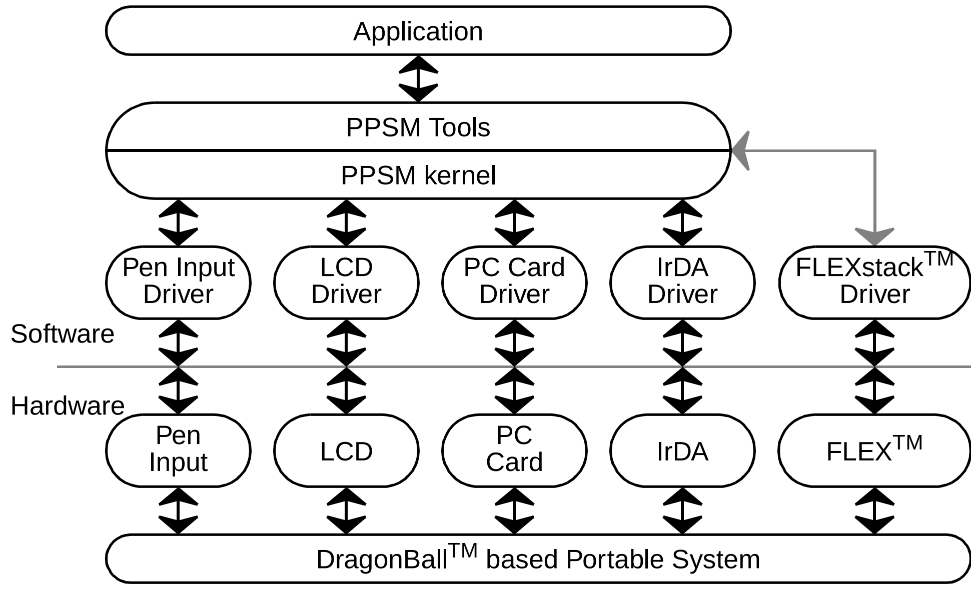


Figure 1-1 Architecture of PPSM on the MC68328 Platform

1.2 Strengths and Features

PPSM is modularly designed to speed the application development cycle time by shielding the developer from the intricacies of the DragonBall™ hardware and providing a good toolset for the application software developer to concentrate on their specific product design. Perfect for handheld devices, PPSM is a resource effective system which requires very small memory space and is pen-centric.

Strengths:

- SPEEDS up application development cycle time
- COMPACT ROM size perfect for handheld devices
- MODULAR Software Architecture
- PEN-CENTRIC
- Third party applications successfully ported on PPSM such as MULTILINGUAL HANDWRITING RECOGNITION, SCALABLE and BITMAP FONTS, Infrared and Email COMMUNICATION.
- Application code is completely DEVICE INDEPENDENT
- Microsoft Windows based ANSI C development tools for RAPID application development

Features:

- Pen / touch panel input support
- 32-bit Real Time Operating System
- Real Time Interrupt Handling
- Power Management
- 16-bit text data representation for multilingual support

- High Level API Toolsets
- Multiple grey levels and software configurable LCD display support
- LCD hardware cursor and panning support

1.3 Software Development Environment

ANSI C is the main programming language, with M68K assembly language for implementation of low level routines, such as interrupt, trap vector initialization and hardware device driver. The development environment is on IBM-compatible PC running under Windows 3.1 and Windows 95 with SingleStep Compiler/ Debugger, from Software Development Systems, Inc. SingleStep provides source level and instruction debugging on the hardware target machine, as well as a simulator running on the host PC.

1.4 Hardware Development Environment

The M68328 and M68EZ328 Application Development System (ADS) is used as the reference development platform throughout this manual. The A/D convertor used is a 10-bit component, giving a resolution of 1024 in X and 1024 in Y direction.

For details on the hardware configuration, please refer to *M68328 ADS User Manual V2.0 and M68EZ328 ADS User Manual V1.1.*

Chapter 2 PPSM System Overview

2.1 Interrupt Handling

An application will always be running under the PPSM at any given time. It will either be the active task, or a suspended task while PPSM is servicing an interrupt. An application is scheduled out under any one of the following conditions:

- Interrupt from pen
- Interrupt from a communication device
- Interrupt from power management
- Interrupt from timer
- Interrupt from messages

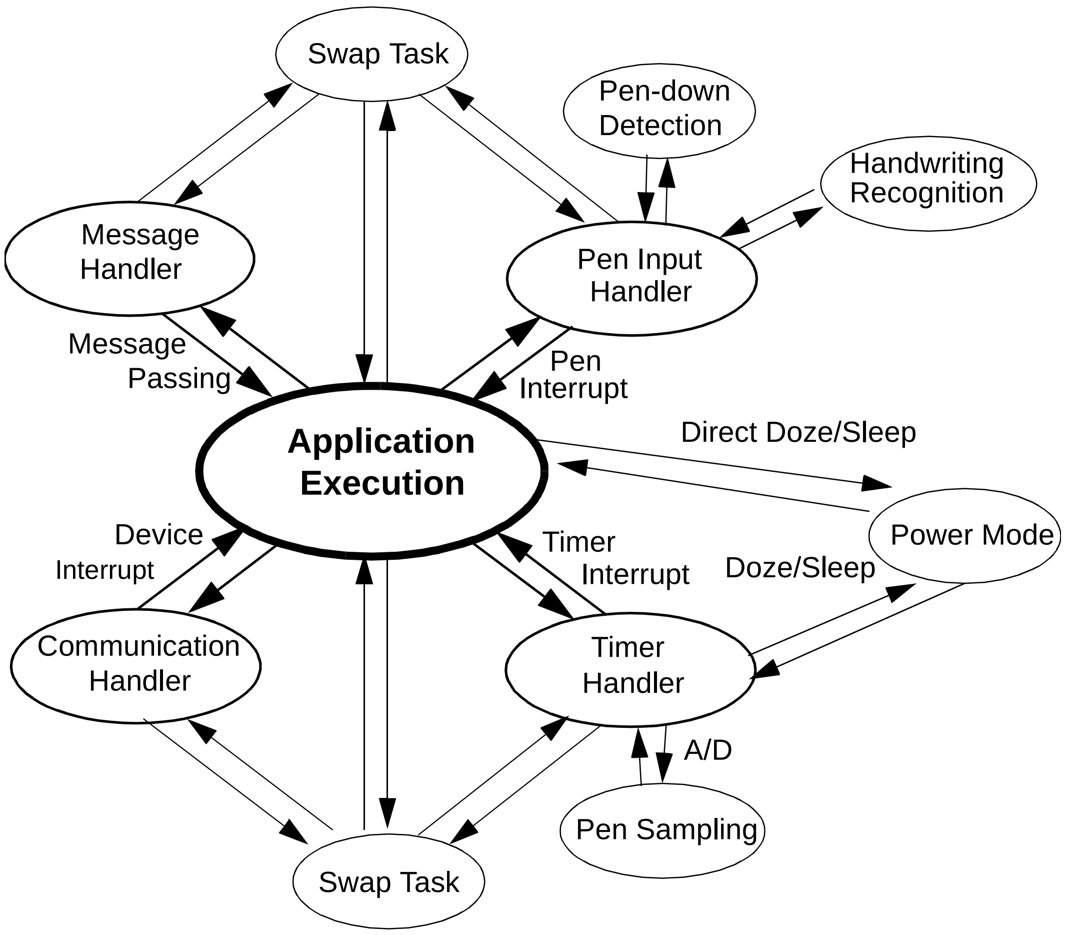


Figure 2-1 Task Execution States

2.2 Error Handling

Unless otherwise specified, all PPSM tools return PPSM_OK upon successful completion. A value other than PPSM_OK is the error code, indicating an error has occurred.

Each error code uniquely defines the cause and nature of the error.

Refer to *Appendix A - Error Code Definition* for a complete error code listing.

2.3 I/O Devices

This section describes the LCD display screen and the touch sensitive panel.

Figure 2-2 shows the screen format for the PPSM system. There are three major areas:

- Pen Input Area
- Display Screen
- Panning Screen

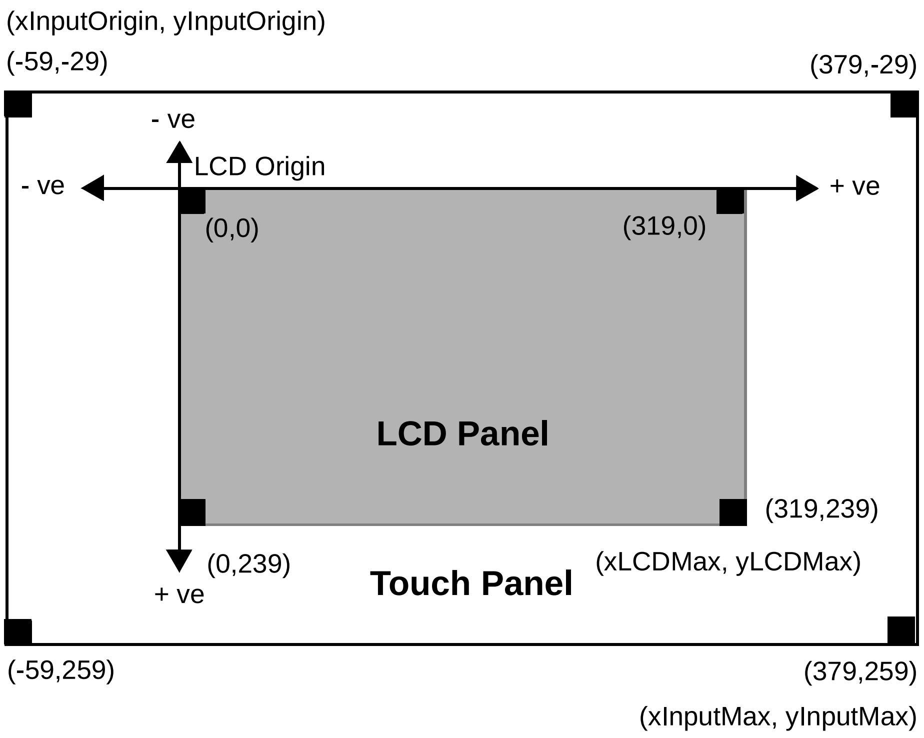


Figure 2-2 An example of PPSM V3.0 coordinate system

2.3.1 Pen Input

2.3.1.1 Pen Input Area

This is the touch sensitive panel input area. The coordinate system used for the touch panel is the same as that for the LCD display screen. The reference point, (0,0) or the origin, is at the top-left corner of the LCD display screen. As you can see from *Figure 2-3*, the input coordinates outside of the LCD display screen can have a negative value. PPSM allows negative coordinates for pen input. This allows applications to implement features such as off screen icon and off screen writing area.

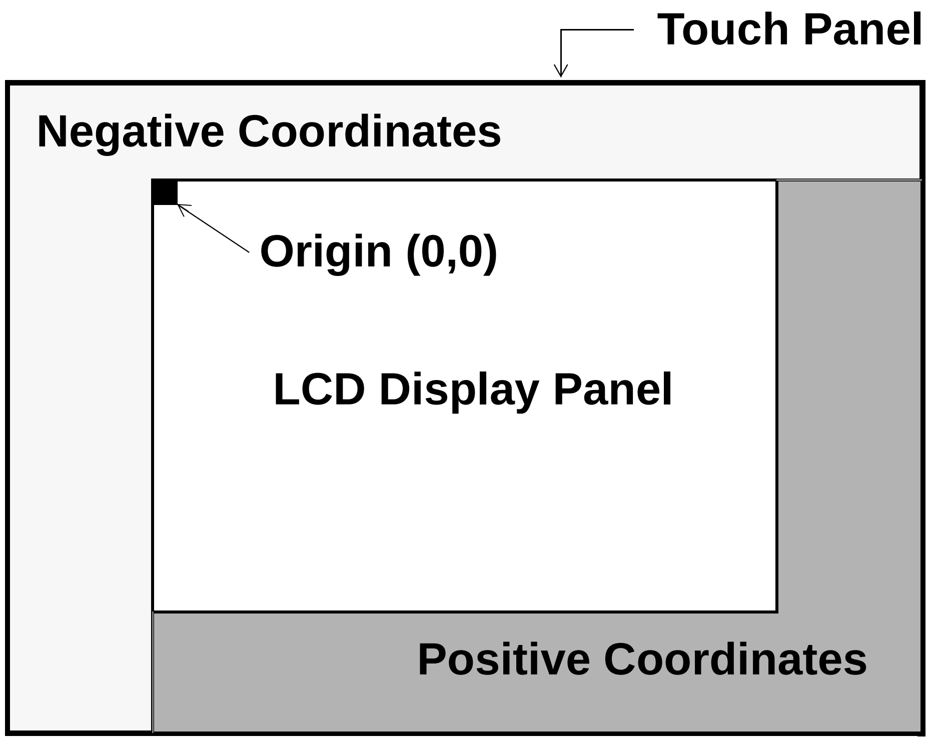


Figure 2-3 PPSM Coordinate System

If the LCD display screen physical size is exactly the same as the touch panel, all coordinates from the pen input will always be positive.

2.3.1.2 Active area

An active area is defined as a rectangular region of the pen input area where an application or an action will execute if the region is pressed. An example of this is an icon, or an action button.

Active areas are classified into two groups, icon area and input area. Please refer to *Section 4.1 - Active Area* for full details on active area definition.

2.3.2 Screen Format

2.3.2.1 LCD Display Screen

The display screen is the LCD display area where applications can display images. The LCD module can handle both 1 bit per pixel and 2 bits per pixel graphics, giving black and white display or 4 grey levels display respectively. Display data, such as graphics and text, can only be seen within the display screen area.

2.3.2.2 Panning Screen

The Panning Screen is an extension to the LCD Display Screen. Its main purpose is to allow applications to write data to an area outside of the actual display area. Although applications can write to this area, data will not be displayed on the screen unless this area is being mapped to the LCD Display Screen. Pen Input areas on the panning screen will receive pen input data only when they overlap with the LCD display screen.

2.3.3 Hardware Cursor

The maximum hardware cursor size is 31 pixels wide by 31 pixels high. During task swapping, the hardware cursor status, size, position and the offset of display origin on panning screen in current task will be saved and the new task□ cursor status, position, size and the offset of display origin on panning screen will be used.

2.4 Data Storage

Databases are the main means of data storage in PPSM. A set of database tools is available in PPSM to support data storage and manipulation.

PPSM database are in global database which can be shared among different tasks.

When a database is created, PPSM will pass back a unique database identifier to the application. Application will need to use this identifier as the key for subsequent access to that particular database. A PPSM database does not have any limit in number of database nor record in database except the memory limitation in creating these database and record.

The PPSM database tools provides basic operations such as add, delete, modify and search for particular data. Additional tools are provided to help manipulate the record list. In particular, the tool DBGetFirstRecID(), DBGetNextRecID(), and DBGetPrevRecID() are meant to facilitate the implementation of more sophisticated searching algorithm. The description, calling convention, and example usage of each database tool will appear in *Chapter 7 - Database Management* and *Chapter 21 - Database Management Tools*

2.5 Font Management

PPSM supports 8x10 English, 16x20 English, 16x16 BIG5 Chinese and 16x16 GB Chinese bitmap fonts, and BIG5 Chinese scalable fonts. The English bitmap fonts are included in PPSM. The other fonts are provided by third party vendors. System integrators need to work with third party vendors about the availability of these fonts.

The font bitmap lookup or generation is handled by a font driver, refer to *Section 33.6 - Font Driver (font.c)*, and is transparent to the applications. Applications gain access to these fonts for displaying text using the PPSM text tools.

2.6 Memory Management

PPSM provides a set of memory management tools for application to access local memory space. A heap is managed by PPSM which allows callers to dynamically allocate memory from the system. When using PPSM, the standard memory tools provided by the compiler will be disabled.

Four memory management tools are available:

`	Lcalloc()	memory allocation with initialization with zero
`	Lmalloc ()	memory allocation
`	Lrealloc()	memory re-allocation
`	Lfree()	memory release

PPSM also provides a set of memory inquiry tools for application to get the run-time memory size.

Three memory inquiry tools are available:

`	TaskMemUsed()	memory used by a task through Lmalloc()
`	TotalMemUsed()	memory used by the whole system through Lmalloc()
`	TotalMemSize()	memory available through Lmalloc()
`	TaskStackAvail()	stack can be used by the current task

The size of the largest chunk of memory can be allocated through Lmalloc() can be known by calling Lmalloc(LARGEST_MALLOC_SIZE).

The size of the heap memory depends very much on the amount of memory available in the hardware system, please refer to *Chapter 32 - How to make ROM?* on how memory size is specified in PPSM.

For SingleStep Debugging System (SDS) user, please refer to *Chapter 34.3* for further description in how to set an optimum size for malloc space.

2.7 Power Management

PPSM utilizes the power control module of DragonBall^{TM} to implement a set of power management tools to achieve system power saving. Applications can choose to control the system’s power management features directly, or use the PPSM’s automatic power management features.

2.7.1 Direct Control

A set of tools provide the applications the ability to directly control the following during Normal mode:

- switch into any of the power saving modes
- the duty cycle of the processor for each application

2.7.2 Automatic Control

A set of tools are available for the caller to set the parameters for automatic power management features provided by PPSM:

- to switch automatically to a lower power saving mode when system is idle
- to control user defined I/O ports during transitions of the power saving modes

2.8 Task Management

Each application running on PPSM is considered as a task. Only one of these tasks can be actively running at anytime.

2.8.1 PPSM Tasks

There are two types of PPSM tasks: application task that are stand alone (main task) and tasks that are spawned off by another task (sub-task).

2.8.1.1 Main Task

Most applications fall into the main task category. Main tasks run independently of each other. There cannot be more than 1 main task running at anytime. They are created by the system tool TaskCreate() or AdvTaskCreate(). Once a main task is created, it can be started in one of the following ways:

- By using the system tool TaskStart()
- By pressing the application icon
- By messages sent by another task

2.8.1.2 Sub-task

Sub-task, on the other hand, can be active at the same time as the parent task that generated the sub-task. Message passing is possible between the sub-task and its parent. Sub-task uses the display resource of its parent and can only be started with the system tool SubTaskCreate().

Sub-tasks are tied to the parent task. If the parent task is swapped out or terminated, the sub-task will be swapped out or terminated too. Sub-task inherits the input pad and panning screen properties from the parent task at creation.

2.8.2 Application State Transition

Application tasks have three states. *Figure 2-4* shows the three states of an application.

- Active
- Suspended
- Stopped

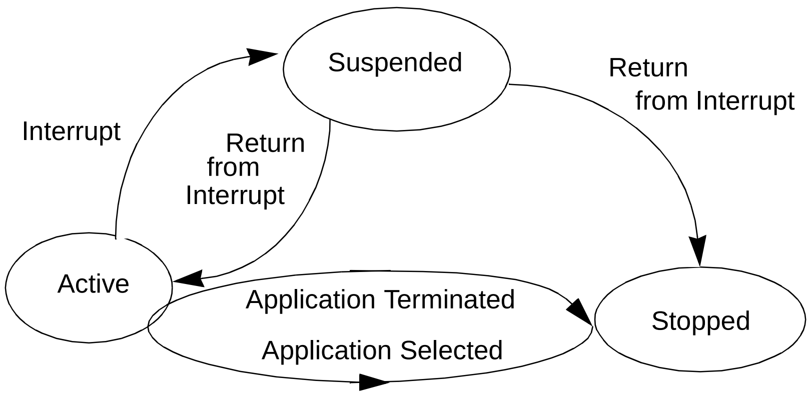


Figure 2-4 Application State Transitions

2.8.2.1 Active State

This is the state when an application is actively being accessed by the user. All hardware resources are available to an active task.

2.8.2.2 Suspended State

An application task is put into suspended state when the kernel is interrupted during active state execution. The interrupt service routine, running in supervisor mode, will become the active task. During the active-to-suspended transition, only the registers that are used by the interrupt handler are saved.

The suspended task will return to active state if no other application is selected, otherwise it will be put into stopped state.

2.8.2.3 Stopped State

An application changes from suspended state to stopped state when another application is selected. In this execution state, all registers and application display bitmap image are stored by the kernel.

A task will exit from the stopped state to active state when the application is reselected. It will continue execution from the point when it was put into stopped state.

2.8.3 Task Swapping

Task swapping is performed by PPSM transparent to the application programmer. A task swapping is executed when an application, other than the actively running application, needs to become active. This normally occurs when PPSM kernel receives an interrupt, such as timer or user pen-down.

For example, when an application icon for a new application is pressed, the task that is actively running will be put into stopped state and the new task will become active.

2.9 Timer Management

PPSM maintains a reprogrammable clock that is defaulted to 9:00am 1st of January, 1997 upon start up. This clock is incremented in 1 second interval and never stops.

A set of timer tools are available for time management. This allows the programmer to set system clock, clock alarm, time-out, and input time-out in their application.

Chapter 3 PPSM Programming

PPSM programming mainly consists of two parts:

- PPSM initialization and applications integration
- Individual PPSM application programming

3.1 PPSM Initialization and Applications Integration

A PPSM system must first be initialized before any PPSM tools and resources can be assigned and used. System integrators may choose to integrate multiple individual PPSM applications into one PPSM system. For example, a simple PPSM organizer may consist of applications such as address book, calendar, scheduler and calculator.

Figure 3-1 shows a typical PPSM system start up flow. First, PPSM must be initialized, then applications to be integrated onto the system must be registered with PPSM individually. The last step to get the system going is to start the application which is chosen to run first.

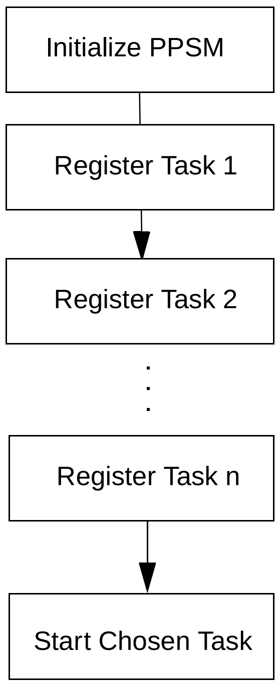


Figure 3-1 PPSM System Start Up Flow Chart

3.1.1 PPSM Initialization

The PPSMInit() tool performs all the internal resources initialization and interrupt handler assignment. It must be called before any other PPSM tool. As an option, it can also perform the pen to screen calibration. This should only be done once at system start-up.

3.1.2 Task Registration

An application is treated as a task in PPSM. After PPSM is initialized, each application task on the system must be registered with PPSM before the application can make use of the PPSM tools.

When integrating the individual applications onto the system, the system integrator must first call one of the task creation tools for each application. There are two types of tasks, main task or sub task (refer to *Section 13.1 - Main Task and Section 13.2 - Sub-task*). This registration of tasks ensures that the run time memory and stack required for each application are allocated within PPSM’s memory system.

Associated with each main application task is an optional application launch icon. This launch icon□ position on the touch panel can be specified in the task creation tool. The application is put to the foreground whenever this application icon is selected by the pen input device.

When writing an application task, the developer can treat each task as a stand alone procedure as PPSM resources are individually allocated. This implementation of tasks allow a number of applications to be written independently and linked together at the end to form a single system.

For details of the task creation tools, please refer to *Section 13.7 - Creating a Task*, *Section 13.8 - Creating a Task with Specific Task Parameters* and *Section 13.9 - Creating a Sub Task*.

After all applications have been registered, the first application task can be started by calling the tool TaskStart(). This tool never returns and other applications within the system will be started when the corresponding application launch icon is pressed.

3.2 PPSM Application Programming

This section describes the general flow for most applications operating under the PPSM environment. The typical flow for most PPSM applications is shown in *Figure 3-2*. After the application initializes itself and registers icons and draw areas with PPSM, it would continuously call IrptGetData() to check for incoming events. When an event occurs, the application would process the event and then loop back to IrptGetData() to wait for more events.

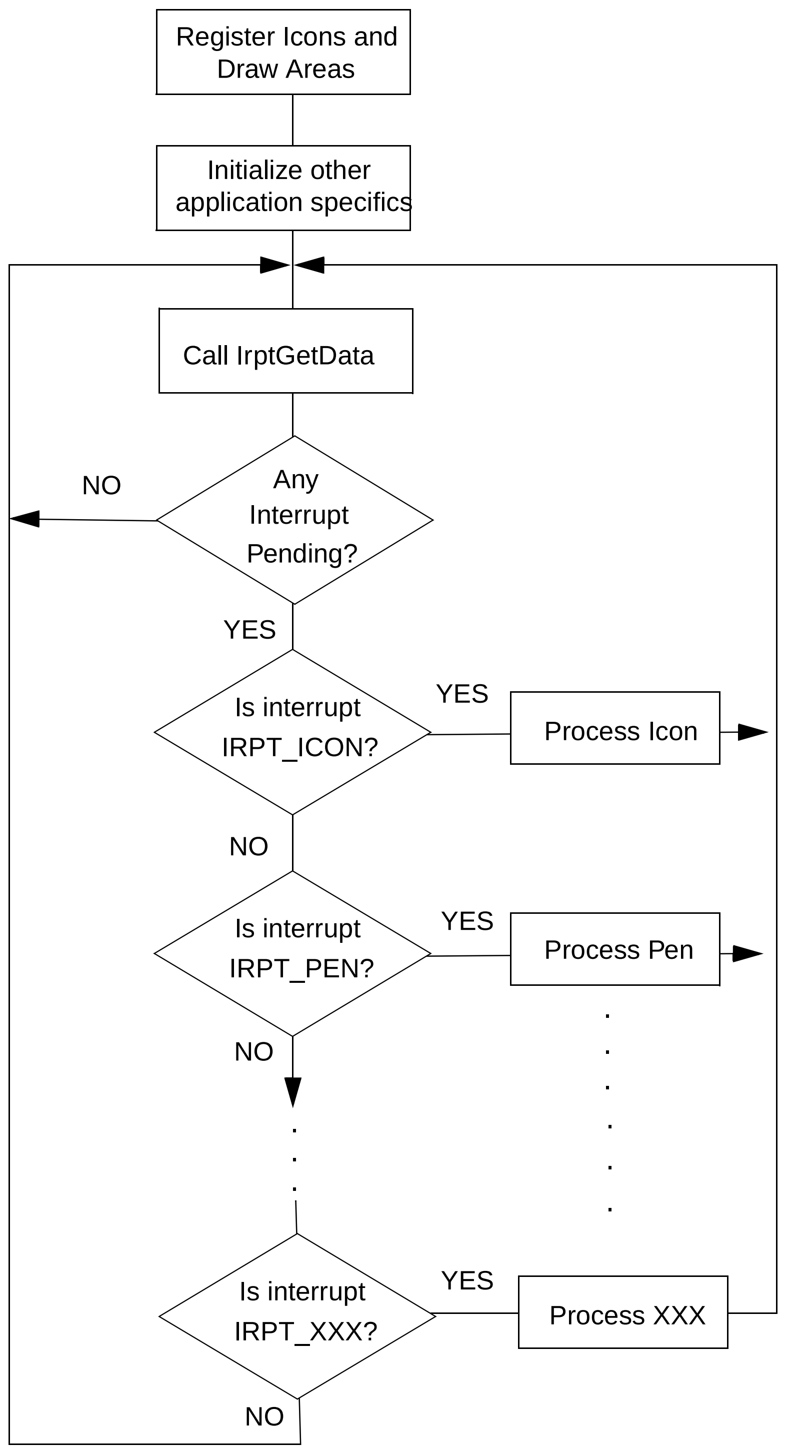


Figure 3-2 Application Flow Chart

3.2.1 Active Area Registration

PPSM is designed as a pen-centric system. Input from the input panel is through regions defined in PPSM as active areas. Active area provides an easy method for applications to receive pen input samples from the input panel without the need to monitor the hardware constantly. PPSM uses interrupt to perform pen sampling, maximizing processor power utilization.

An active area is defined as a rectangular region of the input panel where interrupt messages are generated to the application when the region is pressed. Active areas only generate messages to the application that created the area. An example of an active area is an icon, an action button, scratch pad or drawing area.

When application needs to perform pen input, it must define the area location and register each active area with PPSM before the area can respond to a pen input. All remaining areas on the input panel not registered with PPSM will not generate any information to the application. For details on active areas and input methods, please refer to *Chapter 4 - Pen Input Handling* and *Chapter 5 - Character Input Methods*.

3.2.2 Messages from PPSM

PPSM application should take a pro-active role, i.e., an event driven approach. When external events occur, such as pressing of the input panel, PPSM system automatically intercepts and interprets the event. If the events require attention from the application, such as an active area being pressed, or incoming data from UART, PPSM will package the data in the pre-defined message format and send to the waiting application□ interrupt buffer (please refer to *Chapter 15 - Interrupt Handling*). However, if the event is not intended for the application, such as pressing of input panel that is not defined as active area, or timeout for going into power saving mode, the event is handled by PPSM internally without any message sent to the application.

Upon receiving soft interrupt messages sent from PPSM, application tasks should act upon the nature of the message. There are a set of pre-defined messages types for different types of interrupts (refer to *Section 15.1 - System Interrupts* and *Section 15.2 - Device Interrupts*).

Application tasks can receive the soft interrupt messages by using the system tool IrptGetData(). The application task should call this tool periodically in the program□ flow to check for any incoming events (refer to *Section 29.1 - IrptGetData*).

3.3 Data Representation

PPSM is a 32-bit system. *Table 3-1* shows the data types used in PPSM

Table 3-1 Data type definition used in PPSM

Data Type	Description	Size (in bytes)
U8	unsigned byte	1
P_U8	unsigned byte pointer	4

Table 3-1 Data type definition used in PPSM

Data Type	Description	Size (in bytes)
S8	signed byte	1
P_S8	signed byte pointer	4
U16	unsigned short	2
P_U16	unsigned short pointer	4
S16	signed short	2
P_S16	signed short pointer	4
U32	unsigned int	4
P_U32	unsigned int pointer	4
S32	signed int	4
P_S32	signed int pointer	4
STATUS	unsigned short	2
TEXT	unsigned short	2
P_TEXT	unsigned short pointer	4
P_VOID	void pointer	4

3.4 Naming Convention

For consistency, a set of guidelines is set up for the naming conventions during software development. This is not a fixed requirement, but a recommendation.

3.4.1 Procedure

All procedure names are in lower case except the first letter of each word within the name will be in upper case.

Example 3-1 Procedure names

` PenInit()
` PagerCheck()
` DrawDot()

3.4.2 Constants and Labels

All constants and labels are in upper case, underscores are permitted.

Example 3-2 Constants and labels

` DISPLAY_MODE
` DEFAULT_MODE

- OK
- UNKNOWN

3.4.3 Local Variables

All local variables start with lower case, with capitalised words in the variable names. There are no underscores between words.

Example 3-3 Local variable names

- xPos
- yPos
- dataLen
- temp
- rate

3.4.4 Global Variables

Same as local variables, except that all global variable names starts with a lower case □?

Example 3-4 Global variable names

- gCurrentTask
- gIrptMask
- gSystemClock

3.4.5 Local Pointer Variables

All local pointer variables start with a lower case □? with capitalised words in the variable names. There are no underscores between words.

Example 3-5 Local pointer variable names

- pSourceAddr
- pDestAddr
- pTaskTable
- pReturnSize

3.4.6 Global Pointer Variables

Same as local pointer variables, except that all global variable names starts with a lower case □p`

Example 3-6 Global pointer variable names

- gpSysList
- gpPhoneBook
- gpSrcMem

Part II Writing PPSM Applications

Chapter 4 Pen Input Handling

The Pen Input Tools enable application to:

- define active areas to control pen input
- control ink echoing

Refer to *Section 6.2.2 - Screen Resolution*.

4.1 Active Area

Active area provides an easy method for applications to receive pen input samples from the touch panel without the need to monitor the hardware constantly. PPSM uses interrupt to perform pen sampling, maximizing processor utilization.

An active area is defined as a rectangular region of the touch panel where interrupt messages are generated to the creator task when the region is pressed. An example of this is an icon, an action button, scratch pad or drawing area.

Active areas are only "active" for the task which the pen down has occurred until the corresponding pen up. For example, a main task created active areas "A" and "B", and one of its sub-task(*Chapter 13 - Task Management*) created active areas "C" and "D". When a user taps on active area "A"(i.e. a pen down on "A"), only active areas "A" and "B", but not "C" nor "D", will receive messages if the pen moves across them. By the same token, if pen down is made on "C", only "C" and "D" will receive messages.

If active areas are overlapping, only the active areas of current task will be able to receive pen messages.

There are two types of active areas, icon area and input area. Input area types have three different modes of operation.

Type	Mode	Description
ICON_AREA	N/A	Icon area has only one mode
INPUT_AREA	STROKE_MODE	Stroke input mode
	CONTINUOUS_MODE	Pen position sampling mode
	CONFINED_MODE	Strokes confined within the area

4.1.1 Icon Area

Icon area is for the purpose of selection only. When an icon area is pressed, either from a pen-down or drag in from another area on the touch panel, PPSM generates a soft interrupt and returns the active area identifier to the application.

Upon release, either by pen-up or drag out of the area into another part of the touch panel, another soft interrupt is sent to the application to notify the user of the event. This type of area is designed for buttons and selection icons.

4.1.2 Input Area

Input area is an area where writing or drawing is performed. Once defined, PPSM will monitor the area with the given pen input characteristics such as sampling rate, pen echoing and pen position sampling. Pen echoing is programmable. Three modes of operation are available for this type of area, STROKE, CONFINED or CONTINUOUS.

4.1.2.1 Stroke Mode

Drawing on STROKE type of input area will produce a list of the x and y coordinate integers to the application at the end of the drawing input sequence, usually with a pen-up. This list consists of all points of that single stroke from the pen-input device. When the pen leaves the active area, or pen-up is detected, then the stroke data ends.

4.1.2.2 Confined Mode

CONFINED mode is very much like STROKE mode excepts that when the pen input moves out of the defined active area, the coordinates for those points outside the region are truncated to the value defined by the boundary of the active area. This means a stroke will not be broken until pen-up is detected.

4.1.2.3 Continuous Mode

Drawing on CONTINUOUS type of input area will continuously produce individual x and y coordinates to the application as the pen moves across the pen input panel. With this type of input, soft interrupt is generated for each individual point. The last input point will be a set of (-1, -1) for pen-up. Developers using this type of area must ensure the soft interrupts are acknowledged as their number can be very significant.

4.2 Creating an Active Area.

STATUS **ActiveAreaEnable**(P _U32 *areaId*, U32 *type*, U32 *mode*, S16 *xSrc*,
S16 *ySrc*, S16 *xDest*, S16 *yDest*)

This tool creates a new active area for reading pen input. It returns an identifier of the new area to the caller. Once created, the new active area identifier will be returned to the application.

The argument type is used to specify whether an icon area or an input area is required. Mode specifies the input mode for input area.

Example 4-1 Create an active area

```
U32   nextWinId;/*  NextWin's id                                   */
  .
```

```
 .
 .
/* Create an active area for the NextWin icon with the coordinates
 * as specified below.
 */

if (ActiveAreaEnable(&nextWinId, ICON_AREA, 0, NEXT_WIN_XSRC,
                  NEXT_WIN_YSRC, NEXT_WIN_XDEST, NEXT_WIN_YDEST)
    != PPSM_OK)
  return PPSM_ERROR;

```

4.3 Removing an Active Area

STATUS **ActiveAreaDisable**(U32 *areaId*)

Removes a valid active area from the application. The argument supplied into this tool must be a valid active area identifier that was generated by the ActiveAreaEnable tool.

Once removed, the region of touch panel that was previously defined by the identifier will no longer respond to pen input.

Example 4-2 Remove an active area

```
U32     backId;
 .
 .
 .
           if ( ActiveAreaDisable(backId) != PPSM_OK )
           return (PPSM_ERROR);
```

4.4 Suspending an Active Area

STATUS **ActiveAreaSuspend**(U32 *areaId*, U32 *flag)*

Activate or deactivate a valid active area. Once an active area has been created, it can be suspended from pen input response at anytime.

To suspend, call this tool with the AREA_SUSPEND flag. Once suspended, the active area no longer sends out interrupt messages to the application when writing in the region.

To re-enable the active area, simply call this tool with the AREA_REENABLE flag.

Example 4-3 Suspend an active area

```
if ((rv = ActiveAreaSuspend( iconId, AREA_SUSPEND))
{      /* error */
       return (rv);
}
```

Example 4-4 Re-enable an active area

```
if ((rv = ActiveAreaSuspend( iconId, AREA_REENABLE))
{      /* error */
       return (rv);
}
```

4.5 Active Area Enquiry

STATUS **ActiveAreaRead**(U32 *areaId*, P_S16 *xSrc*, P_S16 *ySrc*, P_S16 *xDest*, P_S16 *yDest)*

Given a valid active area identifier, this tool will return to the caller the coordinates of the active area.

Example 4-5 Enquire coordinates of the active area

```
U32     id, size;
P_U16   inData;
S16     xSrc, ySrc, xDest, yDest;
.
.
.
           ActiveAreaRead(id, &xSrc, &ySrc, &xDest, &yDest);
```

4.6 Put Active Area to Front of List

STATUS **ActiveAreaToFront**(U32 *areaId*)

Given the active area identifier, this tool will extract the element from the active area linked list and insert the element at the front of the list.

Once the element is at the front of the list, it will be the active area to receive pen input if there are other active areas that overlaps the same physical area.

4.7 Pen Echoing

STATUS **AreaEchoEnable**(U32 *areaId*)

STATUS **AreaEchoDisable**(U32 *areaId*)

For input active areas, echoing can be disabled. By default, ink echoing is enabled when input active areas are created. The argument to both tools must be valid active area identifier generated from ActvieAreaEnable tool.

4.8 Pen Color and Pen Size

STATUS **PenEchoParam**(U16 *echoCol,* U16 *echoWidth*)

PPSM allows the application developer to set the echoing pen width and echo pen color. This tool only sets the echoing property of the calling task, and will not affect other applications within the system.

4.9 Creating a Control Active Area

STATUS **CtrlIconEnable**(P_U32 *iconId*, S16 *xSrc,* S16 *ySrc,* U16 *iconType*)

A pre-defined group of direction control icons are available in PPSM for the purpose of scrolling. They are basically icon areas with specific bitmaps mapped to the icon area. Two sets of icons are defined; 8x8 icons and 16x16 icons.

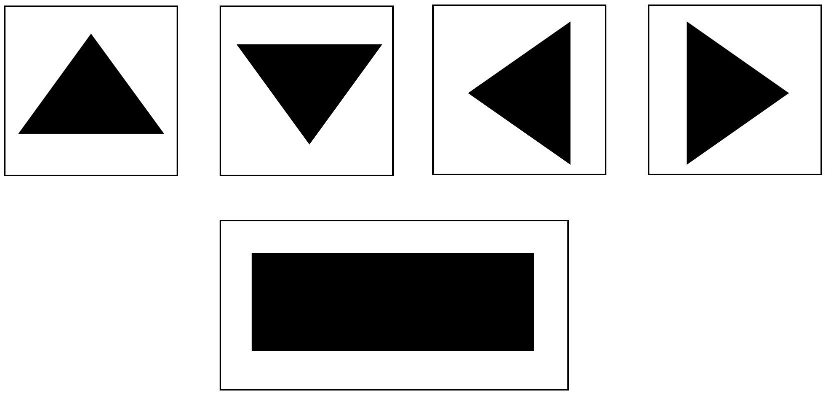


Figure 4-1 Control Icon bitmaps

An identifier is returned to the caller for the new control icon.

Icon Type	Description
PPSM_ICON_8_UP	8 x 8 icon with up arrow bitmap
PPSM_ICON_8_DOWN	8 x 8 icon with down arrow bitmap
PPSM_ICON_8_LEFT	8 x 8 icon with left arrow bitmap
PPSM_ICON_8_RIGHT	8 x 8 icon with right arrow bitmap
PPSM_ICON_8_DONE	8 x 16 icon with rectangle bitmap
PPSM_ICON_16_UP	16 x 16 icon with up arrow bitmap
PPSM_ICON_16_DOWN	16 x 16 icon with down arrow bitmap
PPSM_ICON_16_LEFT	16 x 16 icon with left arrow bitmap
PPSM_ICON_16_RIGHT	16 x 16 icon with right arrow bitmap
PPSM_ICON_16_DONE	16 x 32 icon with rectangle bitmap

Example 4-6 Create a control active area

```
U32        sendButton, rcvButton, abortSend, abortRcv;
.
.
.
/* create control buttons with UP/DOWN ARROW and DONE */
if ( CtrlIconEnable(&sendButton, BUTTON_X, BUTTON_Y, PPSM_ICON_16_UP)
    != PPSM_OK )
  return (PPSM_ERROR);

if ( CtrlIconEnable(&abortSend, BUTTON_X+20, BUTTON_Y, PPSM_ICON_16_DONE)
    != PPSM_OK )
  return (PPSM_ERROR);

if ( CtrlIconEnable(&rcvButton, BUTTON_X, BUTTON_Y+30, PPSM_ICON_16_DOWN)
    != PPSM_OK )
  return (PPSM_ERROR);
```

```
if ( CtrlIconEnable(&abortRcv, BUTTON_X+20, BUTTON_Y+30, PPSM_ICON_16_DONE)
    != PPSM_OK )
  return (PPSM_ERROR);
```

4.10 Removing a Control Active Area

STATUS **CtrlIconDisable**(U32 *iconId*)

Remove the control icon from PPSM. This will remove the icon and the icon will no longer generate icon interrupt to the application.

4.11 Push Active Area List into Background

STATUS **ActiveListPush**(void)

PPSM maintains a stack for storing active area lists. When this tool is called, all the active areas that have been created so far will be pushed into background, and a new list begins. A new list has no element by default. Subsequent active area created will belong to the new active list.

This tool allow applications to create a temporary active area list, and return to the original active areas when the temporary list is no longer required.

Example 4-7 Push active area list into background

```
/* Push all existing active areas to background */
If (rv = ActiveListPush())
{      /* error */
       return (rv);
}

/* create new set of active areas */
GenerateNew();

/* Call subroutine */
ScratchPad();

/* restore original active areas */
if (rv = ActiveListPop())
{      /* error */
       return (rv);
}
```

4.12 Pop Active Area List to Foreground

STATUS **ActiveListPop**(void)

PPSM maintains a stack for storing active area lists. When this tool is called, the least recently pushed in active area list becomes the current active area list. This tool must be called after an ActiveListPush() call has been made previously. Otherwise, an error will be returned.

Chapter 5 Character Input Methods

PPSM supports two types of input methods for applications to receive character input from the user. There is a soft keyboard for typed English character and numeric input, and an input pad for handwritten character input. The input pad will support any coded language that the handwriting recognition engine supports.

This chapter describes the mechanism of the input methods and the Character Input Tools provided by PPSM to support them.

5.1 Soft Keyboard

A default QWERTY soft keyboard with key size of 15x15 pixels can be opened at any position within the panning screen. There are two soft keyboard layouts: one for upper case letters and numbers (refer to *Figure 5-1*); the other for lower case letters and symbols (refer to *Figure 5-2*). The user can toggle between the two layouts by pressing one of the SHIFT buttons.

1	2	3	4	5	6	7	8	9	0
Q	W	E	R	T	Y	U	I	O	P
A	S	D	F	G	H	J	K	L	BACK
SHIFT	Z	X	C	V	B	N	M		RET

Figure 5-1 Upper Case Soft Keyboard Layout

!	@	#	$	&	;	:	*	,	.
q	w	e	r	t	y	u	i	o	p
a	s	d	f	g	h	j	k	l	BACK
SHIFT	z	x	c	v	b	n	m		RET

Figure 5-2 Lower Case Soft Keyboard Layout

As an alternative, an user may define its own keyboard with required number of column and row in number of keys, key width and height in number of pixels, user defined return keycode and the bitmap needed to be used for the soft keyboard.

For both of the above keyboards, together with the keycode, the coordinate (S16, S16) of the pen on the key is also returned to users.

Only one soft keyboard can be opened within each task.

5.1.1 Starting Soft Keyboard Character Input

STATUS **OpenSoftKey**(U16 *xPos*, U16 *yPos*)

OpenSoftKey() opens the soft, or pseudo, keyboard input module. A soft keyboard (as shown in *Figure 5-1* above) is drawn on the panning screen, with its upper-left corner position specified by the caller. When this function is called, PPSM saves the display area covered by the soft keyboard and monitors the input keys automatically. The soft keyboard is now ready for user input.

When the user presses a key on the soft keyboard, the ASCII code for that key will be returned to the calling application by way of IRPT_KEY messages when the application calls IrptGetData(). (Refer to *Section 29.1 - IrptGetData*). One IRPT_KEY interrupt message is generated for each key pressed by the user. The ASCII code returned is of type TEXT, i.e. 2-byte format with zero extended in high byte.

Example 5-1 Open soft keyboard for input

```
/* open soft keyboard for input */
if ( OpenSoftKey(KEYBD_X, KEYBD_Y) != PPSM_OK )
return (PPSM_ERROR);
```

STATUS **AdvOpenSoftKey**(U16 xPos, U16 yPos, U16 keyWidth, U16 keyHeight, U16 numCol, U16 numRow, P_U16 keyMap, P_U8 bitmap)

AdvOpenSoftKey() opens the soft keyboard input module with advanced configurable details.

- Location of the soft keyboard
- Width and height of the keys in number of pixels
- Number of rows and columns of keys
- The return code of each key (keycode)
- The bitmap user interface for the soft keyboard

The return codes are defined in an array(keymap). The order of the keys in the array is from top left key across to the right, then next row and so on, until the bottom right key. The contents of this array should not be changed after AdvOpenSoftKey() has been called. The bitmap has to be either a NULL pointer or it must fit to cover the entire soft keyboard area. Hence, the width and height of this soft keyboard must be (keyWidth*numCol) and (keyHeight*numRow) in number of pixels. For the NULL pointer case, it will not draw anything on the screen.

Example 5-2 Open soft keyboard for input

```
/*  7, 8, 9, 4, 5, 6, 1, 2, 3, *, 0, # */
static const U16 keyMap[] = {55, 56, 57 ,52, 53, 54, 49, 50, 51, 42, 48, 35};
```

```
/* open user specified soft keyboard for input like below */
/* with 10x10 key size and 3 col. x 4 rows. */
/*  7 8 9  */
/*  4 5 6  */
/*  1 2 3  */
/*  * 0 #  */
if ( AdvOpenSoftKey(KEYBD_X, KEYBD_Y, 10, 10, 3, 4, (P_U16)keyMap, bitMap) != PPSM_OK )
    return (PPSM_ERROR);
```

5.1.2 Auto-Key-Repeat

Continually pressing a key would result in auto-key-repeat. The time between the first and second returned key is called AUTO_REPEAT_LIMIT. It is currently set to be 20. The time duration between returned keys after the second one is called AUTO_REPEAT_RATE. It is 5. They are actually the number of pen sampling tick. For example, a 32Hz pen sampling would result in the second key after 20/32 sec, then, for every 5/32 sec each.

5.1.3 Terminating Soft Keyboard Character Input

STATUS **CloseSoftKey**(void)

CloseSoftKey() closes the soft keyboard which is opened by either OpenSoftKey() or AdvOpenSoftKey(). PPSM will restore the display area that was covered by the soft keyboard automatically.

5.1.4 Suspend Soft Keyboard Character Input

STATUS **ActiveListPop**(void)

STATUS **ActiveListPush**(void)

ActiveListPush() pushes the current active area list and the soft keyboard of the current task into background. ActiveListPop() pops the top background active list and soft keyboard of the current task from the active area stack. The active list that is currently being used is destroyed, replaced by the top background active list. For more details, please see *Section 4.1 - Active Area*.

Example 5-3 Display characters received from soft keyboard

```
if (rv = OpenSoftKey(50, 50))
{      /* error */
       return (rv);
}
while (running)
{
       switch(IrptGetData((P_U32)&id, (P_U32*)&inData, (P_U32)&size))
       {
              /* Any key pressed */
              case IRPT_KEY:
                     /* User writing in soft keyboard */
                     DisplayKey((U16) (size >> 1), (P_U16)inData);

                     /* (x, y) store the position of the key being pressed */
                     x=*(++inData);
                     y=*(++inData);
                     DisplayXY(x, y);
```

```
                    break;
                /* Close keyboard icon selected */
                case IRPT_ICON:
                    if (id == CloseIconId)
                        CloseSoftKey();
                        break;
                /* no more interrupt */
                default:
                    break;
            } /* switch */
        } /* while */
```

5.2 Handwriting Recognition Input Pad

The handwriting recognition input pad consists of a number of square boxes in a row by column format layout (refer to *Figure 5-3*).

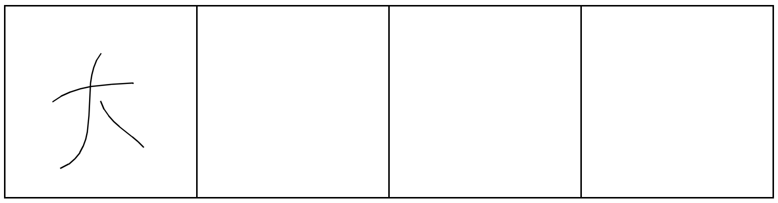


Figure 5-3 An Example Input Pad with 1 row by 4 columns layout

It serves as an interface between the user and the underlying handwriting recognition engine. It captures the stroke data generated from the user□ handwriting input, and passes these data to the handwriting recognition engine for processing. (Refer to *Figure 5-4* for the flow of input and output data passing through the input pad).

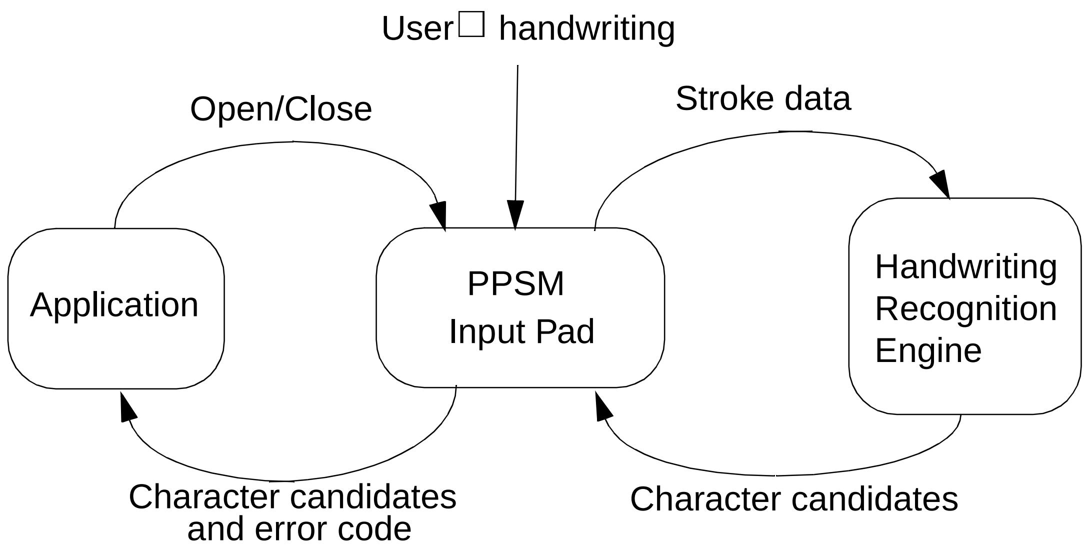


Figure 5-4 Data flow of Handwriting Recognition Input

5.2.1 The Input Pad Mechanism

The user writes a character within an input box to input it into the system. The system proceeds to recognize a character when the user starts writing in a different box, or when a predefined time has passed since the user lifts the pen, whichever occurs first. The characters are recognized in the order they are entered, independent of the box location. The application must define its own mechanism to determine when character input is finished and close the input pad (e.g. the user clicks on a close input pad button created by the application).

The input pad is a subtask. Once it is opened, handwriting recognition interrupt messages, IRPT_HWR, are generated to its main task when characters are being recognized. Each individual recognized character generates an individual IRPT_HWR interrupt message. Through this interrupt message, the system returns the recognized character candidates to the main task which opened the input pad. (Refer to *Section 15.1.8 - IRPT_HWR*).

Only ONE instance of the input pad is supported per main task(ie. shared among parent and sub-tasks). If the user opened an input pad in one of the tasks, an attempt to open the input pad in any other task in its parent/sub-tasks chain would fail. If the user switches to another task chain while an input pad has been opened in the current task. The handwriting recognition engine is reset and the input box area is cleaned if the *areaClean(see below)* flag has been set to be 1 (TRUE).

5.2.2 Starting Handwriting Character Input

STATUS **OpenInputPad**(U16 *xPos*, U16 *yPos*, U16 *numRow*, U16 *numCol*,
U16 *areaSize*)

An application can open the input pad anywhere within the panning screen. The application needs to specify the xy-coordinate of the upper left corner of the input pad, the number of rows and columns of input boxes, and the size of each input box (in units of pixel). If the input pad is not already opened by another application and that the specified layout fits within the panning screen, it will be displayed at the specified location ready for user□ input.

The area of the panning screen covered by the input pad is saved at the time this function is called. Any changes to this covered area by the application after this function is called will not be recorded by the system.

The default length of timeout for OpenInputPad() after the last stroke is 1sec.

STATUS **AdvOpenInputPad**(U16 xPos, U16 yPos, U16 numRow, U16 numCol, U16 areaWidth, U16 areaHeight, U16 echoCol, U16 echoWidth, U32 timeOut, U16 samplingRate, U8 areaClean, U16 stackSize)

AdvOpenInputPad is similar to the tool OpenInputPad but with advanced configurable details. It allows the caller to specify:

` position of the input pad
` number of rows and columns of input boxes
` the width and the height of the input boxes
` the echo ink colour and dot width
` the length of time out after a stroke(no more than 1sec)
` the sampling rate of the pen
` if the system should clean the input box for the user after each character is written
` the stack size for the input pad subtask

5.2.3 Terminating Handwriting Character Input

STATUS **CloseInputPad**(void)

CloseInputPad() closes the input pad that has been opened either by OpenInputPad() or AdvOpenInputPad(). After it is closed, no more handwriting recognition messages will be generated from the system to the application. The original image covered by the input pad is restored by the system if the *areaClean* flag has been set to be 1(TRUE) before.

Example 5-4 Display characters receive from Input Pad

```
/* open an input pad at location (50, 50) that has 1 row of 4 boxes with the boxes
      being 64x64 pixels each */
if (rv = OpenInputPad(50, 50, 1, 4, 64))
{     /* error */
      return (rv);
}
while (running)
{
      switch(IrptGetData((P_U32)&id, (P_U32*)&inData, (P_U32)&size))
      {
            /* Any icon pressed or handwritten character input */
```

```
			case IRPT_HWR:
			/* Code here to handle the IRPT_HWR interrupt to receive the
		recognized character candidates from the system */
			DisplayKey((U16) (size >> 1), (P_U8)inData);
			break;

		/* Close keyboard icon selected */
		case IRPT_ICON:
			if (id == CloseIconId)
				CloseInputPad();
			break;
		/* no more interrupt */
			break;
	} /* switch */
	} /* while */
```

Chapter 6 Using Graphics Tools

PPSM supports LCD modules with capabilities such as multiple grey levels, hardware cursor, hardware panning and software configurable display size.

The Graphics Tools enable applications to:

` draw lines and shapes (e.g. dotted line, circle, etc.)
` display and manipulate bitmap images
` swap bitmap images
` control and restore bitmap images
` control hardware cursor
` set grey levels
` get information about the LCD display screen and panning screen.
` change the panning screen size
` direct all drawing effect to memory area apart from panning screen area
` hardware cursor in inverse and/or blinking mode
` allocate memory for panning screen
` set dot width in pixels for drawing dot, line, rectangle, circle, ellipse, arc and vector
` set pattern fill mode for drawing rectangle, circle, ellipse and arc
` draw vector by connecting consecutive points in a list

This chapter will describe the display screen format, graphics routines and hardware cursor control provided by PPSM.

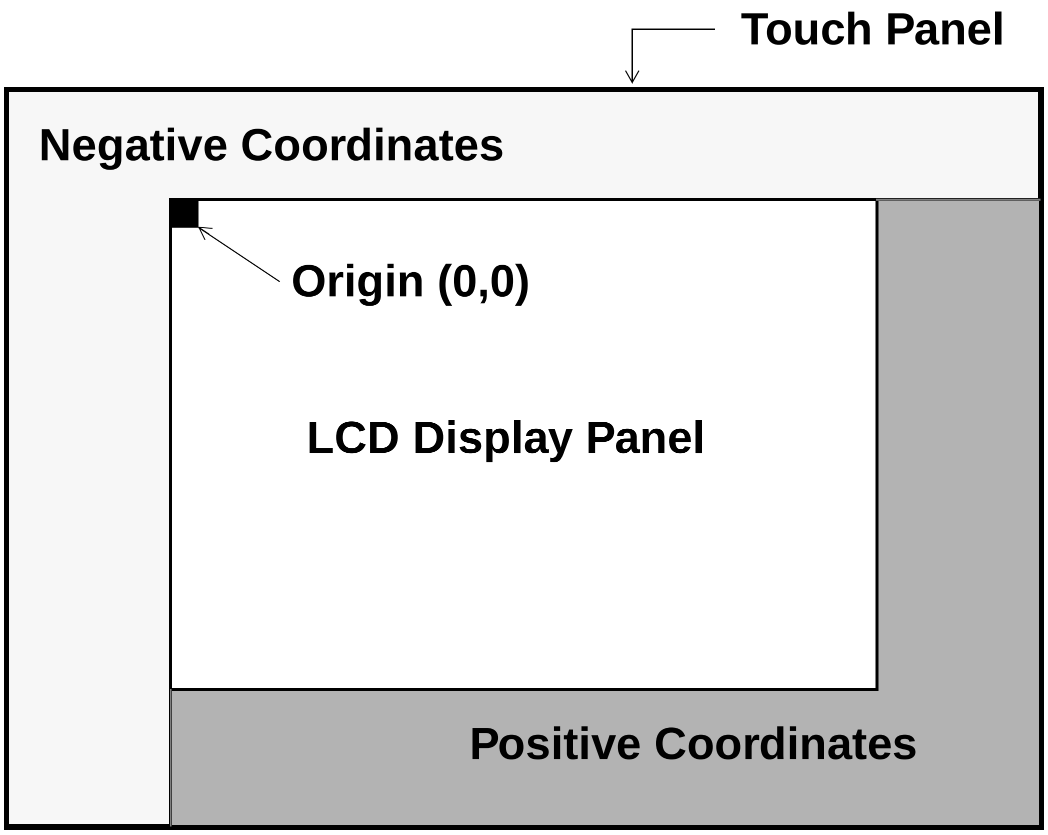


Figure 6-1 Generic Screen Format

6.1 Display Screen Format

Figure 6-1 shows the general screen architecture for the PPSM system. There are three major areas:

- Pen Input Area
- LCD Display Screen
- Panning Display Screen

6.1.1 LCD Display Screen

The Display Screen is the region of the LCD display where applications can display output data. Its size will depend on manufacturer, e.g. 320 pixels wide by 200 pixels high or 320 pixels wide by 240 pixels high, etc.

The LCD module is capable of 1 bit per pixel or 2 bits per pixel output, giving 2 grey levels or 4 grey levels respectively. Hence, there are WHITE and BLACK for 1 bit per pixel and WHITE, LIGHT_GREY, DARK_GREY and BLACK for 2 bits per pixel.

The reference coordinate point for the LCD Display Screen is at the top left corner, the Display Origin. This has the values of (0, 0) initially. Coordinates associated with the LCD Screen are referred to as the display coordinates.

6.1.2 Panning Display Screen

The Panning Screen is an extension to the LCD Display Screen. Its main purpose is to allow applications to write data to an area outside of the actual display area. Although applications can write to this area, data will not be displayed on the screen unless this area is being mapped to the LCD Display Screen. Pen Input areas on the panning screen will receive pen input data only when they overlap with the LCD display screen.

Panning Screen has a similar coordinate system as the LCD Display Screen with different origin. It is configurable but limited by the following:

- The maximum memory size for a panning screen is limited to 64K byte.
- In 1 bit per pixel mode, the width of the panning screen must be divisible by 16 and limited to a maximum of 4096 pixels. In 2 bits per pixel mode, the width must be divisible by 8 and limited to a maximum of 2048 pixels.
- The height of the display is only limited by the amount of memory that is available for a given width.

6.2 Screen Initialization

Because every hardware system will not be identical, a screen initialization procedure is required when PPSM is first activated. This is essential for the calibration of the alignment between the pen input device and LCD display screen.

Screen initialization is implemented when PPSMInit() is called.

Example 6-1 Initialize screen through PPSMInit()

```
main()
{
.
.
.
  /* Initialize PPSM with pen calibration */
  PPSMInit(TRUE);
```

6.2.1 LCD Display Screen in relation to the Touch Panel

The touch panel can be larger than the LCD Display Screen. However, the PPSM tools will only return LCD display screen coordinate. When a pen is touching outside the LCD display region, the display coordinate returned may be negative or even greater than the display screen size.

6.2.2 Screen Resolution

LCD display screen resolution is the number of pixels that are available on the hardware. This is normally a fixed figure for each LCD hardware panel. The touch panel resolution must be equal to or greater than the LCD resolution or the pen cannot point at every pixel of the LCD display.

6.3 Sample LCD Display Screen

The following two figures show example systems used by the M68328ADS hardware platform.

The LCD Display Screen area has two sizes, 320 pixels wide by 240 pixels high and 320 pixels wide by 200 pixels high. The touch panel resolution is 1024x1024.

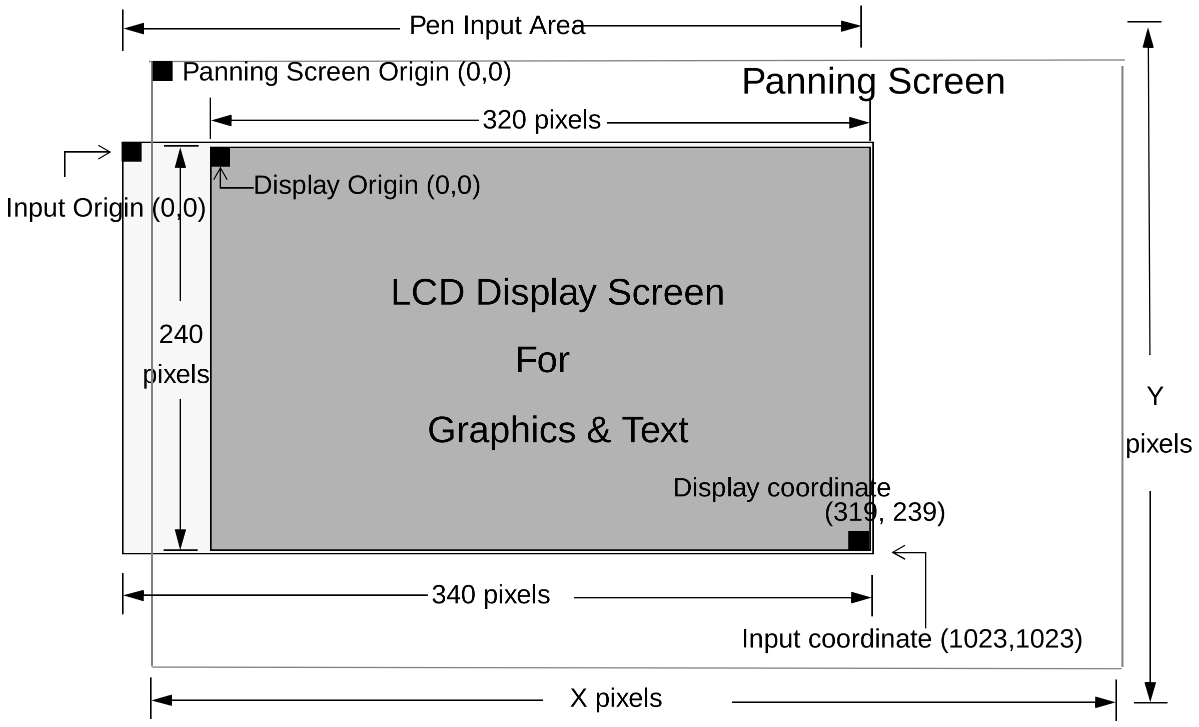


Figure 6-2 320x240 LCD Panel with a larger touch panel

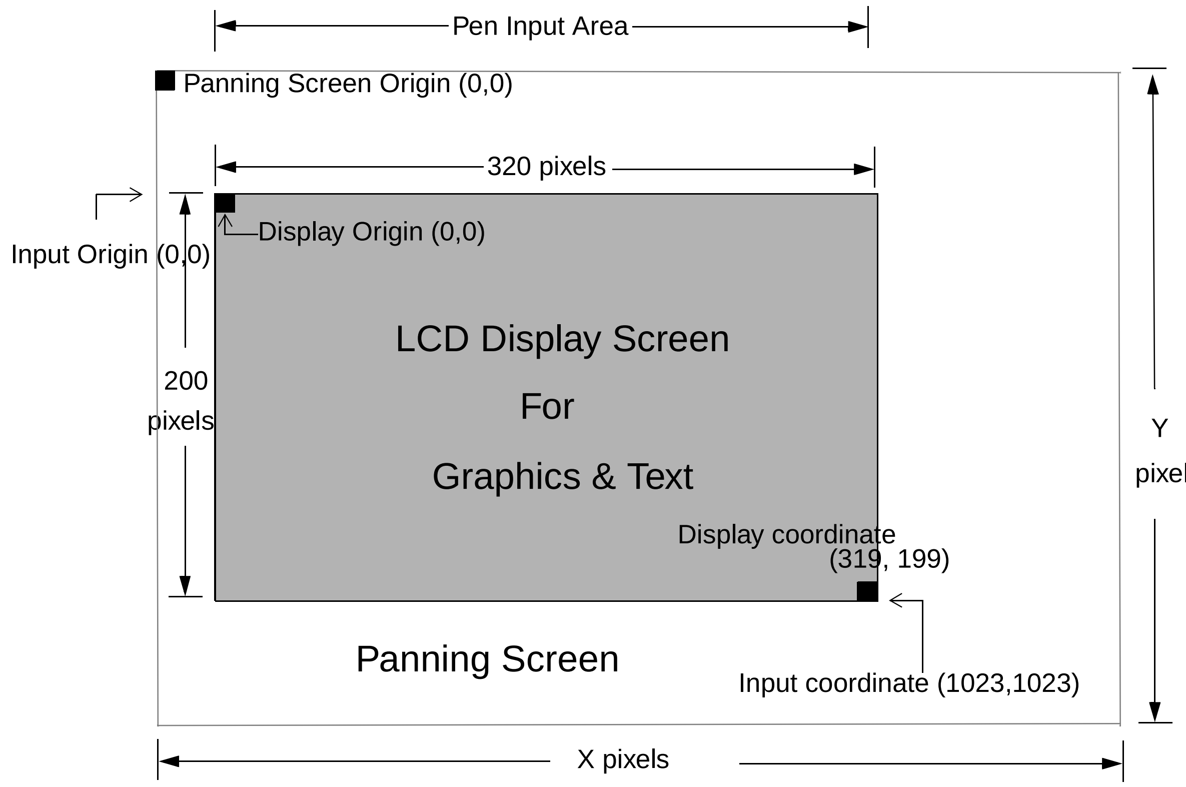
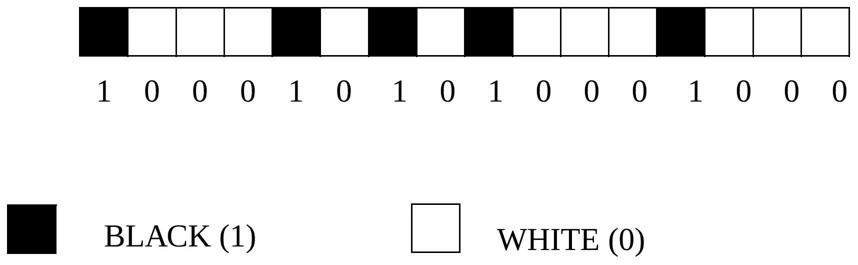


Figure 6-3 320x200 LCD Panel with a same size touch panel

6.4 1 bit-per-pixel Graphics

The graphics routines will handle drawing of BLACK and WHITE pixels on the panning screen.

The byte and bit within byte ordering are both in big-endian format. For example:



The above image will be represented by 1000101010001000 in binary and 0x8A88 in hexadecimal.

6.4.1 Drawing Operators

PPSM supports drawing operators in DrawDot(), DrawHorz(), DrawVert(), DrawLine(), DrawCircle(), DrawEllipse(), DrawArc(), DrawVector(), DrawRec(), PutRec(), and ClearRec().

The following tables show the grey level result of a pixel after a drawing operator is applied.

X is the existing grey level on the screen. Y is the grey level to be put on screen and R is the final grey level on screen after implementation.

AND_STYLE

Table 6-1 R = X AND Y

X	Y	R
0	0	0
1	0	0
0	1	0
1	1	1

OR_STYLE

Table 6-2 R = X OR Y

X	Y	R
0	0	0
1	0	1
0	1	1
1	1	1

EXOR_STYLE

Table 6-3 R = X EXOR Y

X	Y	R
0	0	0
1	0	1
0	1	1
1	1	0

REPLACE_STYLE

Table 6-4 R = Y

X	Y	R
0	0	0
1	0	0
0	1	1
1	1	1

INVERT_STYLE

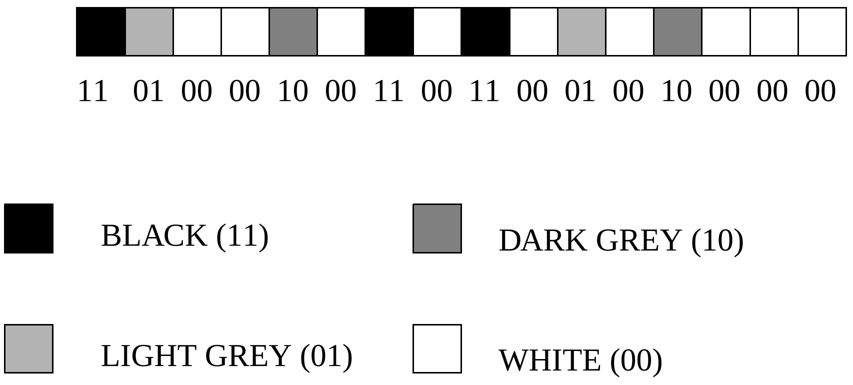
Table 6-5 R = NOT X

X	R
0	1
1	0

6.5 2 bits-per-pixel Graphics

The graphics routine will handle drawing of 4 grey levels: WHITE, LIGHT GREY, DARK GREY and BLACK.

The byte and bit within byte ordering are both in big-endian format. For example:



The above image will be represented by 11010000100011001100010010000000 in binary and 0xD08CC480 in hexadecimal.

6.5.1 Drawing Operators

PPSM supports drawing operators in DrawDot(), DrawHorz(), DrawVert(), DrawLine(), DrawCircle(), DrawEllipse(), DrawArc(), DrawVector(), DrawRec(), PutRec(), and ClearRec().

The following tables show the grey level result of a pixel after a drawing operator is applied.

X is the existing grey level on the screen. Y is the grey level to be put on screen and R is the final grey level on screen after implementation.

AND_STYLE

Table 6-6 R = X AND Y

X	Y	R
00	00	00
01	00	00
10	00	00
11	00	00
00	01	00
01	01	01
10	01	00
11	01	01
00	10	00
01	10	00
10	10	10
11	10	10
00	11	00
01	11	01
10	11	10
11	11	11

OR_STYLE

Table 6-7 R = X OR Y

X	Y	R
00	00	00
01	00	01
10	00	10
11	00	11
00	01	01

Table 6-7 R = X OR Y

X	Y	R
01	01	01
10	01	11
11	01	11
00	10	10
01	10	11
10	10	10
11	10	11
00	11	11
01	11	11
10	11	11
11	11	11

EXOR_STYLE

Table 6-8 R = X EXOR Y

X	Y	R
00	00	00
01	00	01
10	00	10
11	00	11
00	01	01
01	01	00
10	01	11
11	01	10
00	10	10
01	10	11
10	10	00
11	10	01
00	11	11
01	11	10
10	11	01
11	11	00

REPLACE_STYLE

Table 6-9 R = Y

X	Y	R
00	00	00
01	00	00
10	00	00
11	00	00
00	01	01
01	01	01
10	01	01
11	01	01
00	10	10
01	10	10
10	10	10
11	10	10
00	11	11
01	11	11
10	11	11
11	11	11

INVERT_STYLE

Table 6-10 R = NOT X

X	R
00	11
01	10
10	01
11	00

6.6 Graphics Tools

The following sections explain each graphics tool with examples.

Top left corner of the panning screen is (0, 0) which is the panning screen origin. Top left corner of the LCD display screen is an offset from the panning screen origin. All co-ordinates given to the graphics routines are referred to the panning screen origin. There is no negative co-ordinate for panning screen.

Some of the graphics routines need to draw dotted line where one of the argument in calling the routine specifying the length of the dot, e.g. 5 means drawing 5 dots with specifying grey level and then skipping 5 dots and so on. Those graphics routines providing dotted line function are DrawHorz(), DrawVert(), DrawLine() and DrawRec().

DrawDot(), DrawHorz(), DrawVert(), DrawLine(), DrawRec(), DrawCircle(), DrawEllipse(), DrawArc(), ClearRec() and PutRec() provides function of *style* which is described above as OR_STYLE, EXOR_STYLE, AND_STYLE, etc.

For all the following examples in this chapter, the panning screen size is 640x480, LCD display screen size is 320x240 and the LCD display origin is at an offset of (50, 50) from panning screen origin.

6.7 Get LCD Display Screen Width

U16 **GetDisplayX**(void)

GetDisplayX() returns to the caller the physical width, in terms of pixels, of the LCD display panel being used.

When writing an application, this routine should be used instead of using specific numbers for the width of the LCD display screen as it will make the code more flexible to run on different LCD panels.

6.8 Get LCD Display Screen Height

U16 **GetDisplayY**(void)

GetDisplayY() returns to the caller the physical height, in terms of pixels, of the LCD display panel being used.

When writing an application, this routine should be used instead of using specific numbers for the height of the LCD display as it will make the code more flexible to run on different LCD panels.

Example 6-2 Get LCD display screen width and height

```
STATUS DrawTextIcon(P_U32 areaId, U16 xSrc, U16 ySrc, U16 width, U16 height,
                 U16 font, P_TEXT message)
{

    U16     xDest, yDest;
.
.
.
    /* Check to see if the coordinates are fall within the LCD screen */
    if ( (xSrc < 0) || (xDest >= GetDisplayX()) || (ySrc < 0) ||
       (yDest >= GetDisplayY()) )
      return PPSM_ERROR;
```

6.9 Get Panning Screen Width

U16 **GetLogicalX**(void)

GetLogicalX() returns to the caller the panning screen width, in terms of pixels, of the current application.

The panning screen width is dynamically configurable at run time and it is recommended to use this tool to obtain the panning screen width for panning purposes.

6.10 Get Panning Screen Height

U16 **GetLogicalY**(void)

GetLogicalY() returns to the caller the panning screen height, in terms of pixels, of the current application.

The panning screen height is dynamically configurable at run time and it is recommended to use this tool to obtain the panning screen height for panning purposes.

6.11 Set Pattern Fill

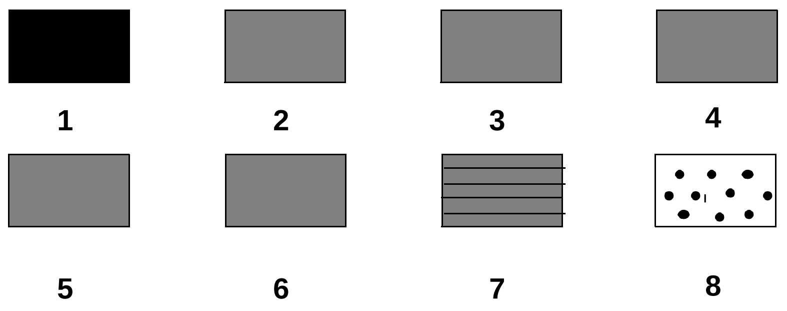
STATUS **SetPatternFill**(U16 *mode*, U16 *backGrey*, U16 *borderMode*, U16 *fillSpace*)

This routine allows application programmers to decide on the fill pattern settings. These settings include the pattern mode, the spacing between the pattern lines, the background grey level, and the existence of a border. Once SetPatternFill() is called, the settings will be applied to all subsequent DrawRec(), DrawCircle(), DrawEllipse(), and DrawArc().

The pattern will be drawn with the specified grey level in the parameter of DrawRec(), DrawCircle(), DrawEllipse(), and DrawArc().

The argument *fillSpace* lets application developers define the size of the gap between the pattern lines. The size of the gap equals to $2^{fillSpace}$ number of pixels.

There are 8 fill patterns available (mode 0 will turn off the pattern fill feature):



The pattern fill mode 0 will turn off the pattern fill feature.

6.12 Set Dot Width

STATUS **SetDotWidth**(U16 *newWidth*, P_U16 *oldWidth*)

After this routine is called, the new dot width will take effect in all subsequent DrawDot(), DrawHorz(), DrawVert(), DrawRec(), DrawLIne(), DrawCircle(), DrawEllipse(), DrawArc(), and DrawVector().

If the dot width is larger than 1, a thick dot, thick line, thick circle, thick ellipse, thick arc and thick vector lines can be drawn.

Example 6-3 Set dot width

```
 * Drawing1 - Draw a LINE and then an ELLIPSE with pattern filled on Screen1.
 *            Both have a dot width = 6, but no border on the ellipse is
 *            drawn as the bordermode in SetPatternFill is set to 0.
    .
    .
    .
  SetDotWidth(6, 0);
  SetPatternFill(2, WHITE, 0, 3);
```

6.13 DisplayMove

STATUS **DisplayMove**(U16 *xPos*, U16 *yPos*)

This function is to set the relative coordinate of top left corner of LCD in panning screen. It sets the display region on LCD from panning screen. Whenever this function is called, the new area in panning screen will be refreshed on LCD.

6.14 Direct All Graphics Output to off-screen memory

STATUS **ChangeWindow**(U32 *addr*, U16 *width*, U16 *height*, P_U32 *oldAddr*, P_U16 *oldWidth*, P_U16 *oldHeight*)

Run-time computation intensive image generation and display could be slow. Users may see the graphics output appears slowly on the LCD display screen. ChangeWindow() allows applications to direct all output from PPSM graphics routines to an off-screen memory area temporarily, so that no changes will appear on the LCD display screen while the image is being built. Once the image is generated, it can be displayed onto the panning screen. This will give the effect that the image is displayed instantaneously.

ChangeWindow() assumes that all input parameters are valid. No zero or invalid values are supposed to be passed as input to this routine.

Example 6-4 Use ChangeWindow() to draw image

```
U32 oldScreen1, oldScreen2;
U16 oldWidth1, oldHeight1;
```

```
U16 oldWidth2, oldHeight2;
U32 tempScreen = (U32)GetScreenMem(64, 64);

/* Direct all graphic routine to memory area pointed by
   tempScreen with width 64 pixels and height 64 pixels */
ChangeWindow(tempScreen, 64, 64, &oldScreen1, &oldWidth1, &oldHeight1);

/* This may be any routine for calculating the co-ordinates to
   be drawn or drawing anything by calling graphics routines */
DoCalculation();

/* Direct all graphic routine back to original panning screen */
ChangeWindow(oldScreen1, oldWidth1, oldHeight1, &oldScreen2, &oldWidth2,
      &oldHeight2);
```

6.15 Change Panning Screen Parameters

STATUS **ChangePanning**(P_PAN_SCREEN *newPanning*, U16 *flag*,
P_PAN_SCREEN *oldPanning*)

ChangePanning() allows applications to change the active panning screen to another memory area during run-time. The *flag* is to indicate whether the old panning screen is still needed or it will be destroyed.

P_PAN_SCREEN is a pointer to structure PAN_SCREEN which has the following elements:

Table 12-11 Panning Screen Parameters

Name	Description
panAddress	The memory address where panning screen origin is located. All graphics routines will draw in the area relative to this address.
horzSize	The panning screen width in number of pixels.
vertSize	The panning screen height in number of pixels.
displayXOrigin	The x-coordinate of the LCD display screen relative to the panning screen. In normal case, when ChangeWindow() is not called, *displayXOrigin* and *displayYOrigin* are used to calculate the value of *displayScreenAddr* and *regPOSR*.
displayYOrigin	The y-coordinate of the LCD display screen relative to the panning screen. In normal case, when ChangeWindow() is not called, *displayXOrigin* and *displayYOrigin* are used to calculate the value of *displayScreenAddr* and *regPOSR*.

Table 12-11 Panning Screen Parameters

Name	Description
displayScreenAddr	The memory address where the LCD display screen origin is located. In normal case, when ChangeWindow() is not called, this is calculated from *panAddress*, *displayXOrigin* and *displayYOrigin*. This is where the system will retrieve the first word to display on LCD display screen. This is used in the Screen Starting Address Register (SSA) described in 4.7.1.1 of MC68328/ MC68EZ328 Integrated Processor User Manual.
regPOSR	This is the pixel offset within the word pointed to by *displayScreenAddr* where the first pixel on LCD display screen will be retrieved. This is used in the Panning Offset Register (POSR) in 4.7.5.5 of MC68328/MC68EZ328 Integrated Processor User Manual.
regPSW	This is the panning screen width in number of words. This is used in the Virtual Page Width Register (VPW) in 4.7.1.2 of MC68328/MC68EZ328 Integrated Processor User Manual.

Upon start-up, *panAddress* equals *displayScreenAddr*, *regPOSR* is 0, *displayXOrigin* and *displayYOrigin* are 0 and *regPSW* equals *horzSize*/8 for a 2 bits per pixel display or *horzSize*/16 for a 1 bit per pixel display.

When DisplayMove() is called, both *displayScreenAddr* and *regPOSR* will be modified so that the system will know from which word and from which pixel position within the word the LCD display screen should start mapping.

When ChangeWindow() is called, *panAddress*, *horzSize* and *vertSize* will be changed so that all graphics routines will use these 3 parameters for calculation and image processing.

If all or several tasks in a system want to share a common panning screen to save memory, the following steps can be followed:

1) In main()
   - call GetScreenMem() to create a memory area for the common panning screen
   - call ChangePanning() to free the PPSM system panning screen
   - for each task that shares the common panning screen, call AdvTaskCreate() with parameter PPSM_NOSCREEN to create the task with no panning screen
2) for each corresponding task that shares the common panning screen
   - during initialization, call ChangePanning() with *flag* parameter

equals to FALSE to copy all properties of the common panning screen to the task□ own context

Example 6-5 Use ChangePanning() to let all tasks share the same panning screen

```
#include <ppsm.h>
#include <errors.h>
#include <proto.h>

PAN_SCREEN newScreen;

STATUS TestApp()
{
  /* use the common panning screen in this task */
  ChangePanning(&newScreen, FALSE, 0);

  /* draw a circle with center at (100, 100) and radius 80 pixels */
  DrawCircle(BLACK, 100, 100, 80, REPLACE_STYLE);

}

main()
 {
  U32 taskId;
  P_U8 newPanning;

  PPSMInit(FALSE);

  /* create a common panning screen with size 640x400 */
  newPanning = (P_U8)GetScreenMem(640, 400);

  newScreen.panAddress = (U32)newPanning;
  newScreen.displayScreenAddr = (U32)newPanning;
  newScreen.horzSize = 640;
  newScreen.vertSize = 400;
  newScreen.displayXOrigin = 0;
  newScreen.displayYOrigin = 0;
  newScreen.regPOSR = 0;
  newScreen.regPSW = 80;/* as the LCD panel used is 2 bits per pixel,
                           PSW = 640/8 = 80. */

  /* destroy the system panning screen and use the new panning screen created above
      */
  ChangePanning(&newScreen, FALSE, 0);

  /* clear whole panning screen */
  ClearScreen(WHITE);

  /* draw a circle with center at (100, 100) and radius 20 pixels to the new panning
      screen */
  DrawCircle(BLACK, 100, 100, 20, REPLACE_STYLE);

  /* create a new task with no panning screen */
  AdvTaskCreate(&taskId, (P_VOID)TestApp, 0, 0, 0, 0, 3048, PPSM_NOSCREEN, 0, 0, 0);

  TaskStart(taskId);
 }
```

In the example above, both the system task in main() and task TestApp() share the same panning screen. So whatever done on the panning screen by the system task or task TestApp() will have effect on the panning screen.

If a panning screen is shared among tasks. Calls ChangePanning() to free the screen in one of the task can also free/corrupt the others?screen memory area.

6.16 Fill the whole Panning Screen

STATUS **ClearScreen**(U16 *greyLevel*)

This routine will fill the whole panning screen with the specified grey level.

Example 6-6 Fill the whole screen with WHITE

```
STATUS ret;

/* fill the whole panning screen with white */
ret = ClearScreen(WHITE);
```

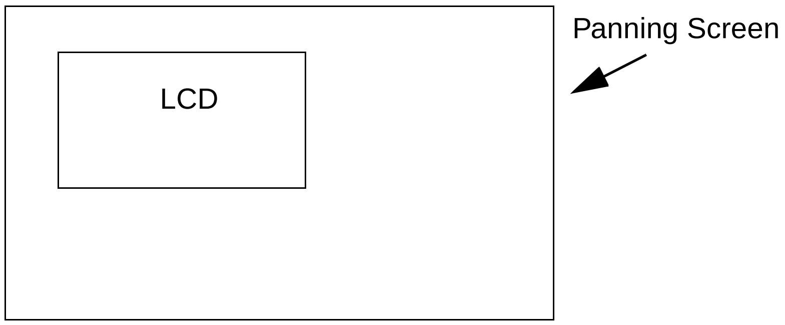


Figure 6-4 Screen output for *Example 6-6*

6.17 Draw a Dot

STATUS **DrawDot**(U16 *greyLevel*, U16 *xPos*, U16 *yPos*, U16 *style*)

This routine will draw a dot at the specified position (*xPos*, *yPos*).

If dot width is 1, a pixel will be drawn. If dot width is 2, a square dot of length 2 will be drawn with top left pixel position as the dot co-ordinate, (*xPos*, *yPos*). When the dot width is greater than 2, a circular disc with radius to be truncated integer value of (dot width - 1)/2 will be drawn. The center of the disc will be the dot co-ordinate, (*xPos*, *yPos*).

Example 6-7 Draw a black dot

```
STATUS ret;

/* draw a dot at (52, 52) */
ret = DrawDot(BLACK, 52, 52, REPLACE_STYLE);
```

The calling of DrawDot(BLACK, 52, 52) will draw a black pixel at (52, 52) in panning screen. As the LCD display screen origin is at (50, 50), the point drawn on screen is at (2, 2) in display co-ordinate. Hence, the expected outcome will have a dot which is very close to the display origin.

Example 6-8 Draw a dot with dot width 2

```
STATUS ret;
```

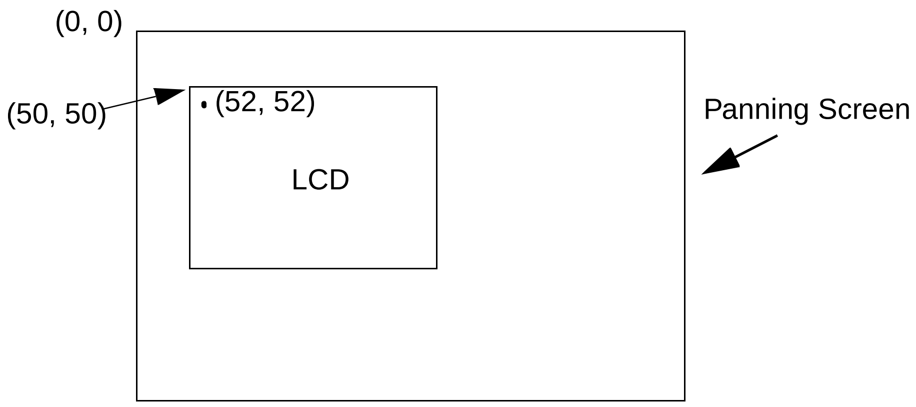


Figure 6-5 Screen output for *Example 6-7*

```
/* set dot width to 2 */
SetDotWidth(2, 0);

/* draw a dot at (52, 52) */
ret = DrawDot(BLACK, 52, 52, REPLACE_STYLE);
```

When the dot width equals 2, a square dot with length of 2 will be drawn.

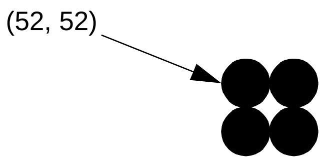


Figure 6-6 Screen output for *Example 6-8*

Example 6-9 Draw a dot with dot width 3

```
STATUS ret;

/* set dot width to 3 */
SetDotWidth(3, 0);

/* draw a dot at (52, 52 */
ret = DrawDot(BLACK, 52, 52, REPLACE_STYLE);
```

When the dot width is 3, a circular disc with radius of (3-1)/2 (which is 1) will be drawn.

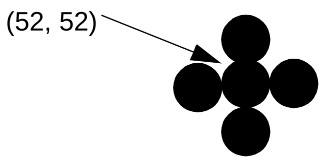


Figure 6-7 Screen output for *Example 6-9*

Example 6-10 Draw a dot with dot width 4

```
STATUS ret;
```

```
/* set dot width to 4 */
SetDotWidth(4, 0);

/* draw a dot at (52, 52) */
ret = DrawDot(BLACK, 52, 52, REPLACE_STYLE);
```

When the dot width is 4, a circular disc with radius of (4-1)/2 (which is 1) will be drawn.

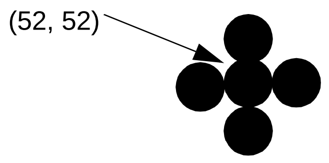


Figure 6-8 Screen output for *Example 6-10*

6.18 Draw a Horizontal Line

STATUS **DrawHorz**(U16 *greyLevel*, U16 *xSrc*, U16 *ySrc*, U16 *width*, U16 *dotLine*, U16 *style*)

This routine will draw a horizontal line from (xSrc, ySrc) to (xSrc + width - 1, ySrc).

If the dot width is greater than 1, the specified horizontal line will have integer truncated of (dot width - 1)/2 horizontal lines above it, and (dot width)/2 horizontal lines below it. The length of each of these lines will be extended by (dotwidth - 1)/2 pixels to the left of the source, and by (dotwidth/2) pixels to the right of the end point.

If the width of the horizontal line is 1, a square dot will be drawn.

Example 6-11 Draw a horizontal black line

```
STATUS ret;

/* draw a black horizontal line from (30, 60) with width 551 */
ret = DrawHorz(BLACK, 30, 60, 551, 0, REPLACE_STYLE);
```

In this example, the dot width is 1. The calling of DrawHorz(BLACK, 30, 60, 551, 0, REPLACE_STYLE) will draw a black horizontal line from (30, 60) to (580, 60) on panning screen. Only the portion of (50, 60) to (369, 60) will be seen on LCD display.

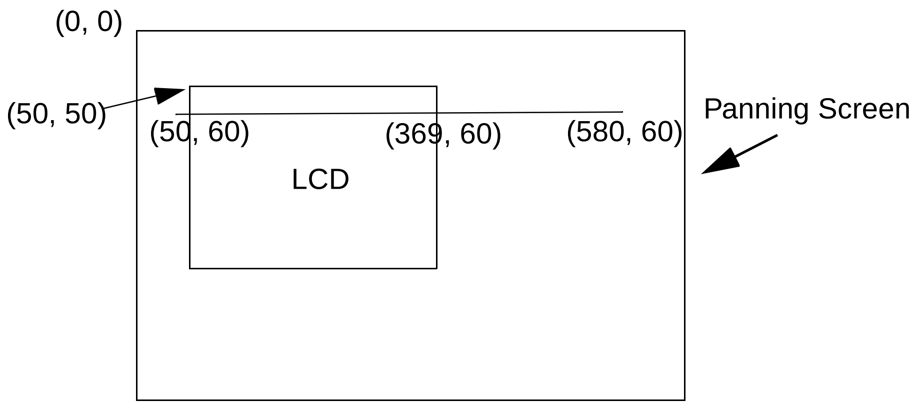


Figure 6-9 Screen output for *Example 6-11*

Example 6-12 Draw a thick horizontal line

```
STATUS ret;

/* set dot width to 4 */
ret = SetDotWidth(4, 0);

if (ret !=PPSM_OK)
return ret;

/* draw a black horizontal line from (60, 60) with width 2 */
ret = DrawHorz(BLACK, 60, 60, 2, 0, REPLACE_STYLE);

if (ret != PPSM_OK)
return ret;
```

In the above example, a thick horizontal line will be drawn as follow:

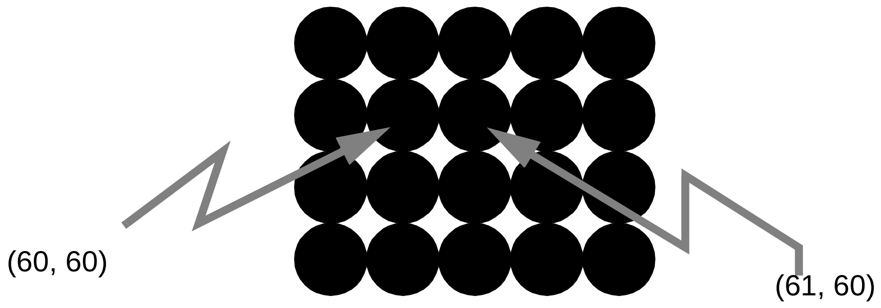


Figure 6-10 Screen output for *Example 6-12*

6.19 Draw a Vertical Line

STATUS **DrawVert**(U16 *greyLevel*, U16 *xSrc*, U16 *ySrc*, U16 *height*, U16 *dotLine*, U16 *style*)

This routine will draw a vertical line from (xSrc, ySrc) to (xSrc, ySrc + height - 1).

If the dot width is greater than 1, the specified vertical line will have integer truncated of (dot width - 1)/2 vertical lines to its left, and (dot width)/2 vertical lines at right. The height of each of these lines will be extended by (dotwidth - 1)/2 pixels above the source and by (dot width)/2 below the end point.

If the height of the vertical line is 1, a square dot will be drawn.

Example 6-13 Draw a vertical black line

```
STATUS ret;

/* draw a black vertical line from (60, 60) with height 361 */
ret = DrawVert(BLACK, 60, 60, 361, 2, REPLACE_STYLE);
```

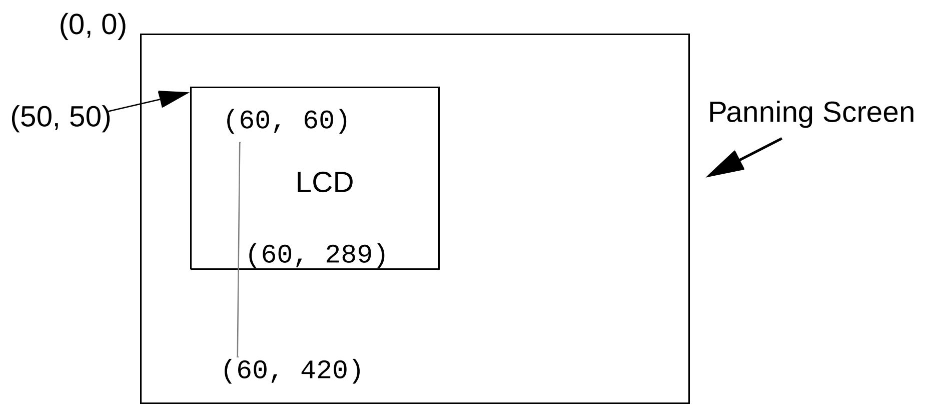


Figure 6-11 Screen output for *Example 6-13*

In this example, the dot width is 1. The calling of DrawVert(BLACK, 60, 60, 361, 0, REPLACE_STYLE) will draw a black line from (60, 60) to (60, 420) on panning screen. However, only the portion of the line on LCD display screen will be seen which is (60, 60) to (60, 289). Since the parameter for dotted line is 2, the line is drawn in the form of 2 BLACK pixels and then 2 WHITE pixels and then 2 BLACK pixels, and so on.

Example 6-14 Draw a thick vertical line

```
STATUS ret;

/* set dot width to 4 */
ret = SetDotWidth(4, 0);

if (ret !=PPSM_OK)
return ret;

/* draw a black thick horizontal line from (10, 10) with height 2 */
ret = DrawVert(BLACK, 10, 10, 2, 0, REPLACE_STYLE);

if (ret != PPSM_OK)
return ret;
```

In the above example, a thick vertical line will be drawn as follow:

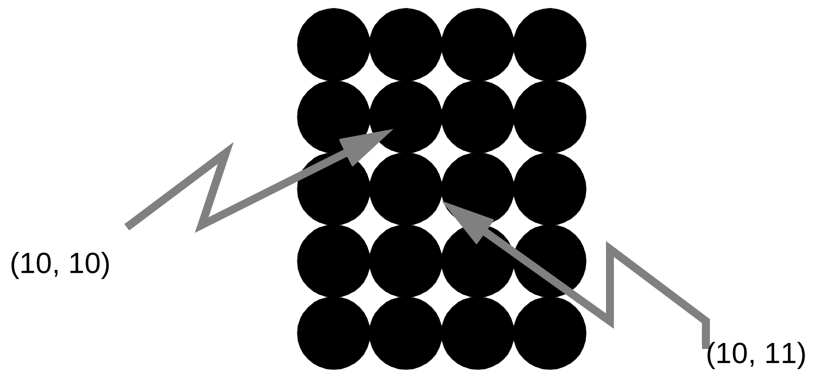


Figure 6-12 Screen output for *Example 6-14*

6.20 Draw a Line

STATUS **DrawLine**(U16 *greyLevel*, U16 *xSrc*, U16 *ySrc*, U16 *xDest*, U16 *yDest*, U16 *dotLine*, U16 *style*)

This routine will draw a line from (xSrc, ySrc) to (xDest, yDest).

If dot width is greater than 1, each dot on the specified line will be represented by a thick square dot. Each of the thick square dot is generated by extending integer truncated (dot width - 1)/2 pixels above, (dot width)/2 below, (dot width - 1)/2 pixels to the left and (dot width)/2 pixels to the right of the original dot. The thick line is then generated by overlapping these thick dots accordingly.

Example 6-15 Draw a black line

```
STATUS ret;

/* draw a black line from (60, 240) to (630, 470) */
ret = DrawLine(BLACK, 60, 240, 630, 470, 0, REPLACE_STYLE);
```

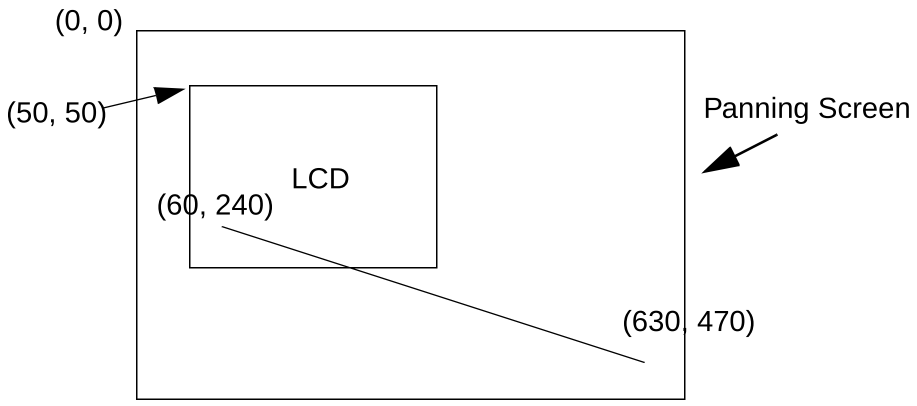


Figure 6-13 Screen output for *Example 6-15*

In this example, the dot width is 1. The calling of DrawLine(BLACK, 60, 240, 630, 470, 0, REPLACE_STYLE) will draw a black line from (60, 240) to (630, 470) on panning screen. However, only the portion of the line on LCD display screen will be seen.

Example 6-16 Draw a thick line

```
STATUS ret;

/* set dot width to 4 */
ret = SetDotWidth(4, 0);

if (ret !=PPSM_OK)
return ret;

/* draw a black thick line from (10, 10) to (11, 11) */
ret = DrawLine(BLACK, 10, 10, 11, 11, 0, REPLACE_STYLE);

if (ret != PPSM_OK)
return ret;
```

In the above example, a thick horizontal line will be drawn as follow:

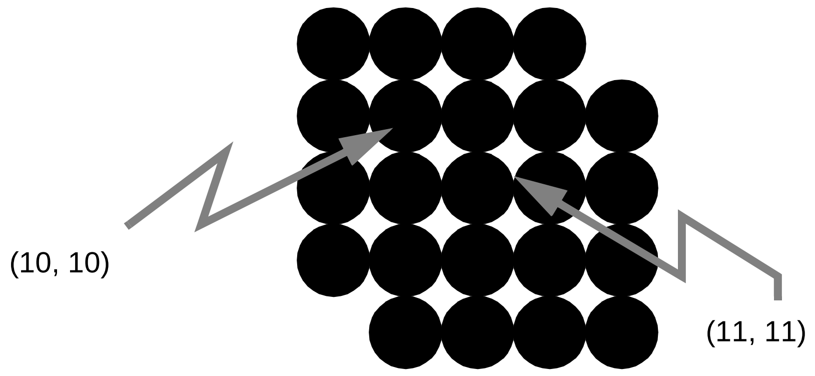


Figure 6-14 Screen output for *Example 6-16*

6.21 Draw a Rectangle

STATUS **DrawRec**(U16 *greyLevel*, U16 *xSrc*, U16 *ySrc*, U16 *xDest*, U16 *yDest*, U16 *dotLine*, U16 *style*)

This routine draws a rectangle with top left corner at (xSrc, ySrc) and bottom corner at (xDest, yDest).

If the dot width is greater than 1, integer truncated (dot width - 1)/2 lines are drawn inside the rectangle and (dot width)/2 lines drawn outside the rectangle.

If both fill pattern mode and border mode are set, those area inside the rectangle which is not covered by the border will be filled.

If fill pattern mode is set and border mode is off, the area inside and on the rectangle border will be filled.

Example 6-17 Draw a rectangle with black outline

```
STATUS ret;

/* draw a black rectangle with top left corner at (310, 250) and bottom right corner
      at (500, 400) */
ret = DrawRec(BLACK, 310, 250, 500, 400, 0, REPLACE_STYLE);
```

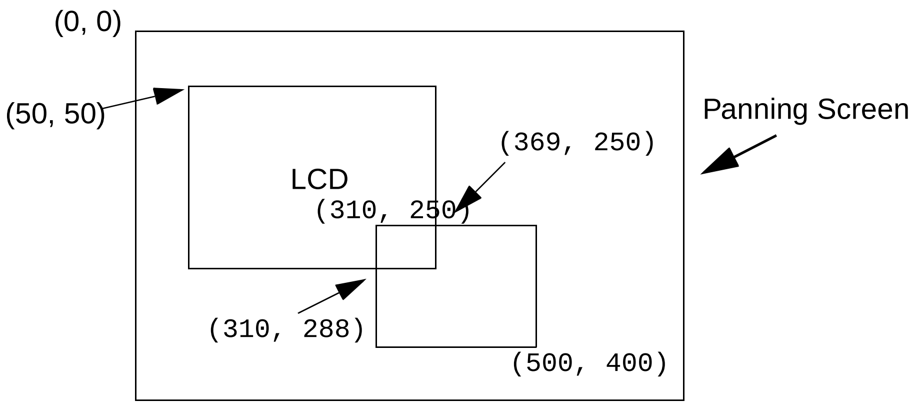


Figure 6-15 Screen output for *Example 6-17*

In this example, the dot width is 1. The calling of DrawRec(BLACK, 310, 250, 500, 400, 0, REPLACE_STYLE) will draw a rectangle with top left corner at (310, 250) and bottom right corner at (500, 400) on panning screen. However, only a horizontal line from (310, 250) to (369, 250) and a vertical line from (310, 250) to (310, 289) will be seen on the LCD display screen.

Example 6-18 Draw a rectangle with black outline in dot width 3 and fill pattern mode 1

```
STATUS ret;

/* set dot width to 3 */
ret = SetDotWidth(3, 0);

if (ret !=PPSM_OK) return ret;

/* set pattern fill mode to 1 which is solid fill */
ret = SetPatternFill(1, WHITE, TRUE, 1);

if (ret !=PPSM_OK) return ret;

/* fill a rectangle from top left corner at (310, 250) to (500, 400) */
ret = DrawRec(BLACK, 310, 250, 500, 400, 0, REPLACE_STYLE);
```

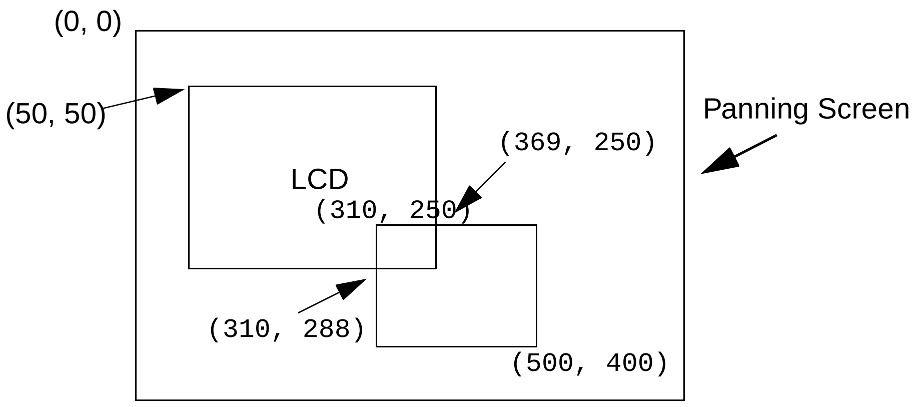


Figure 6-16 Screen output for *Example 6-17*

In this example, the dot width is 3 and fill Pattern mode is 1. The calling of DrawRec(BLACK, 310, 250, 500, 400, 0, REPLACE_STYLE) will fill a rectangle with top left corner at (309, 249) and bottom right corner at (501, 401) on panning screen. However, only a smaller rectangular area from (309, 249) to (369, 289) will be seen on the LCD display screen.

6.22 Draw a Circle

STATUS **DrawCircle**(U16 *greyLevel*, U16 *xCenter*, U16 *yCenter*, U16 *radius*, U16 *style*)

This routine will draw a circle centering at (xCenter, yCenter) with the specified radius and grey level.

If the dot width is greater than 1, integer truncated (dot width - 1)/2 lines are drawn inside the circle and (dot width)/2 lines drawn outside the circle.

If both fill pattern mode and border mode are set, those area inside the circle which is not covered by border will be filled.

If fill pattern mode is set and border mode is off, the area inside and on the circle border will be filled.

Example 6-19 Draw a circle with black outline

```
STATUS ret;

/* draw a black outlined circle with center at (560, 290) and radius 150 */
ret = DrawCircle(BLACK, 560, 290, 150, REPLACE_STYLE);
```

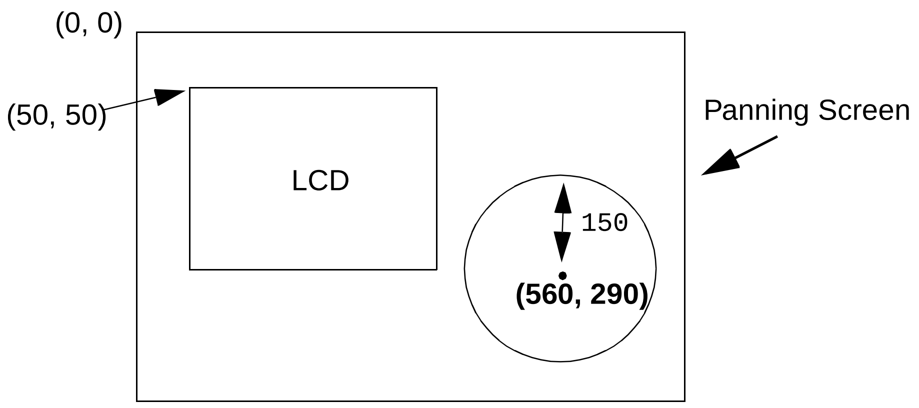


Figure 6-17 Screen output for *Example 6-19*

In this example, the dot width is 1. The calling of DrawCircle(BLACK, 560, 290, 150, REPLACE_STYLE) will draw a circle centering at (560, 290) with radius 150. As the circle is drawn outside the LCD display screen, nothing will be seen on the LCD.

6.23 Draw an Ellipse

STATUS **DrawEllipse**(U16 *greyLevel*, U16 *xCenter*, U16 *yCenter*, U16 *xLength*, U16 *yLength*, U16 *style*)

This routine will draw a ellipse centering at (xCenter, yCenter) with the specified size.

If the dot width is greater than 1, integer truncated (dot width - 1)/2 lines are drawn inside the ellipse and (dot width)/2 lines drawn outside the ellipse.

If both fill pattern mode and border mode are set, those area inside ellipse which is not covered by the border will be filled.

If fill pattern mode is set and border mode is off, the area inside and on the ellipse border will be filled.

Example 6-20 Draw an ellipse with black outline

```
STATUS ret;

/* draw an ellipse with center at (560, 290), horizontal length 150 and vertical
      length 100 */
ret = DrawEllipse(BLACK, 560, 290, 150, 100, REPLACE_STYLE);
```

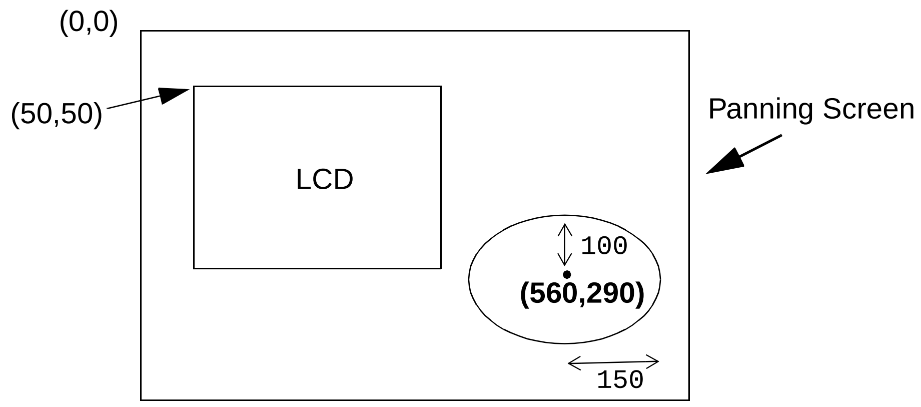


Figure 6-18 Screen output for *Example 6-20*

In this example, the dot width is 1. The calling of DrawEllipse(BLACK, 560, 290, 150, 100, REPLACE_STYLE) will draw an ellipse centering at (560, 290) with the longest distance on y axis from center to border is 100 pixels and the longest distance on x axis from center to border is 150 pixels.

6.24 Draw an Arc

STATUS **DrawArc**(U16 *greyLevel*, U16 *x1*, U16 *y1*, U16 *x2*, U16 *y2*, U16 *style*)

This routine will draw an arc connecting (x1, y1) and (x2, y2).

DrawArc() will draw a quarter of an ellipse centering at (x2, y1). If DrawArc(BLACK, x1, y1, x2, y2, REPLACE_STYLE) is called, the following arcs will be drawn according to the values of (x1, y1) and (x2, y2):

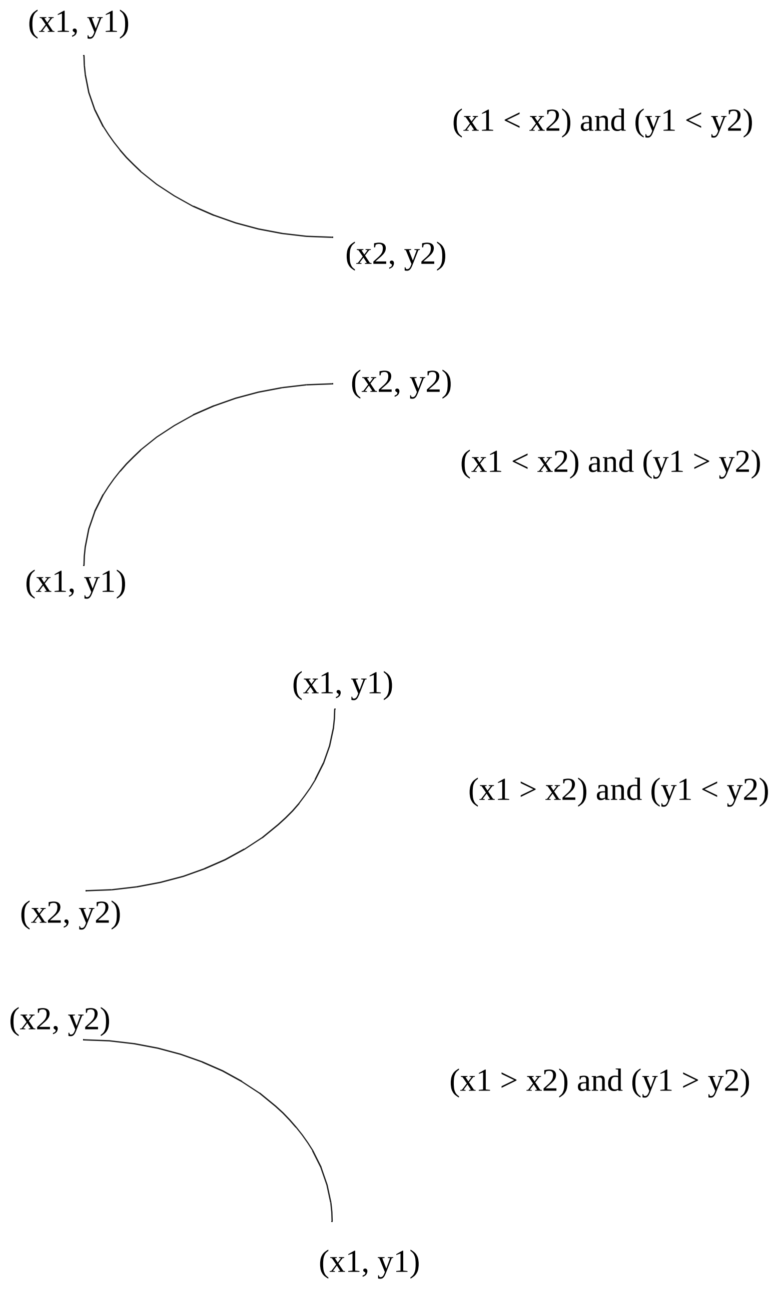


Figure 6-19 Cases of DrawArc

If the dot width is greater than 1, integer truncated (dot width - 1)/2 lines are drawn inside the arc and (dot width)/2 lines drawn outside the arc.

If both fill pattern mode and border mode are set, those area inside arc which is not covered by the border of the arc will be filled.

If fill pattern is set and border is off, those area inside and on the arc border will be filled.

Example 6-21 Draw a black arc with OR style

```
STATUS ret;

/* draw an arc from (240, 150) to (100, 100) */
ret = DrawArc(BLACK, 240, 150, 100, 100, OR_STYLE);
```

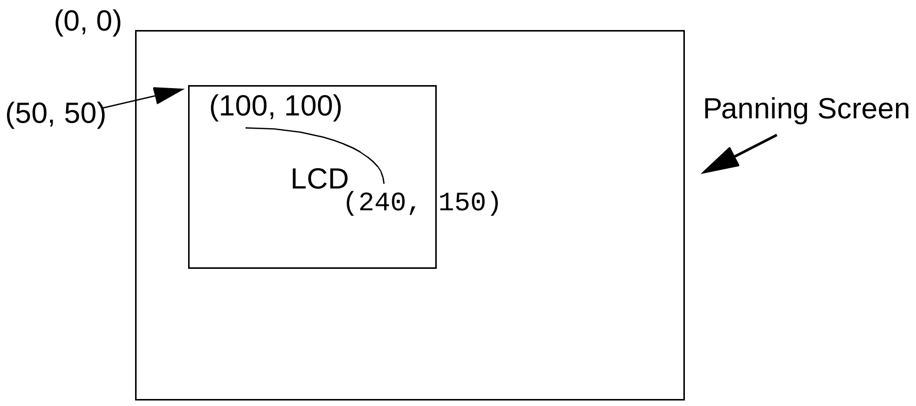


Figure 6-20 Screen output for *Example 6-21*

In this example, the dot width is 1. The calling of DrawArc(BLACK, 100, 100, 50, 50, OR_STYLE) will draw an arc from (100, 100) to (240, 150) on panning screen. The arc is actually a quarter of an ellipse centering at (100, 150) with the longest distance of 141 pixels in x axis and the longest distance of 51 pixels in y axis. The center is determined by the x axis value of the second point and the y axis value of the first point which is 100 and 150 respectively.

Example 6-22 Draw a black arc with EXOR style

```
/* draw an arc from (100, 100) to (240, 150) */
ret = DrawArc(BLACK, 100, 100, 240, 150, EXOR_STYLE);
```

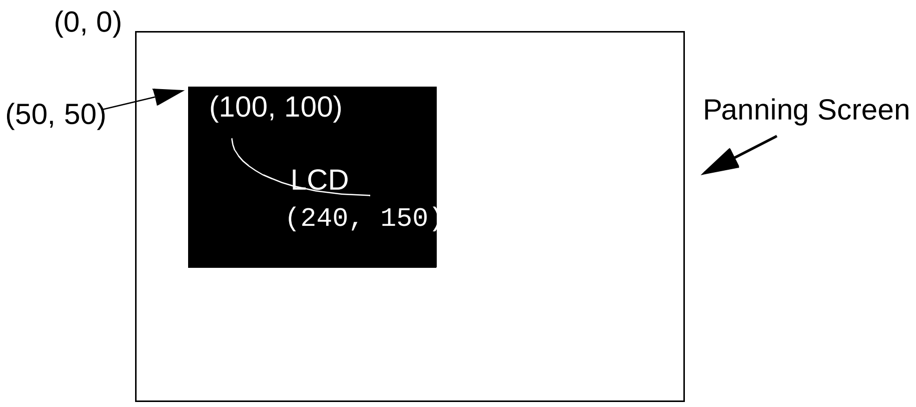


Figure 6-21 Screen output for *Example 6-22*

In this example, the dot width is 1. As the LCD display screen is all BLACK and the calling of DrawArc() is in exclusive OR style, the arc turns out to be WHITE on a black background.

6.25 Draw a Vector from a List of Points

STATUS **DrawVector**(U16 *greyLevel*, U16 *numberOfPoints*, P_POINT *pointPtr*, U16 *style*, U16 *mode*)

This routine will draw lines to connect the points in the given list. It has options on whether the first point and last point need to be connected.

6.26 Put a Rectangular Area on Panning Screen

STATUS **PutRec**(P_U8 *bitmap*, U16 *xSrc*, U16 *ySrc*, U16 *width*, U16 *height*, U16 *style*, U16 *reserved*)

This routine puts an image from memory to panning screen.

PutRec() supports style such as REPLACE_STYLE, OR_STYLE, EXOR_STYLE and AND_STYLE. There is error checking done on the argument *style*, but not on the value of *bitmap.*

Example 6-23 Put a bitmap on screen with REPLACE_STYLE

```
/* put an image on panning screen with top left corner at (0, 0), width 640 and
      height 480 */
ret = PutRec(bitmap, 0, 0, 640, 480, REPLACE_STYLE, 0);
```

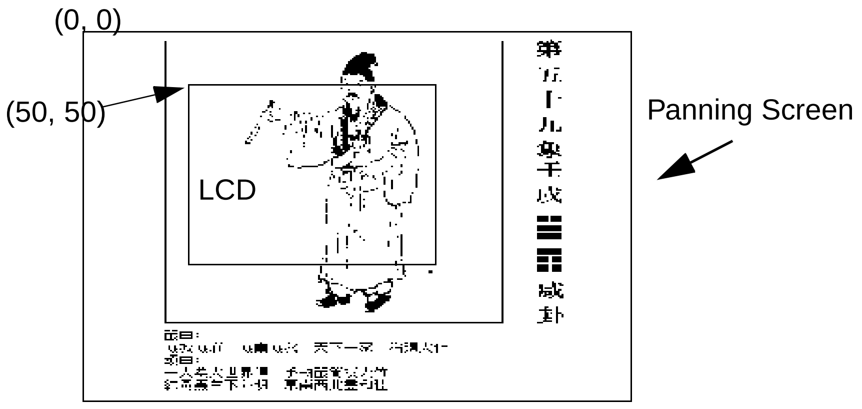


Figure 6-22 Screen output for *Example 6-23*

The calling of PutRec(bitmap, 0, 0, 640, 480, REPLACE_STYLE, 0) copies the image from the memory area pointed to by *bitmap* onto the panning screen.

6.26.1 Special cases of PutRec()

The following are the few special cases of PutRec().

Example 6-24 Display one bit per pixel image on 2 bits per pixel LCD setting

Two similar images will be seen horizontally.

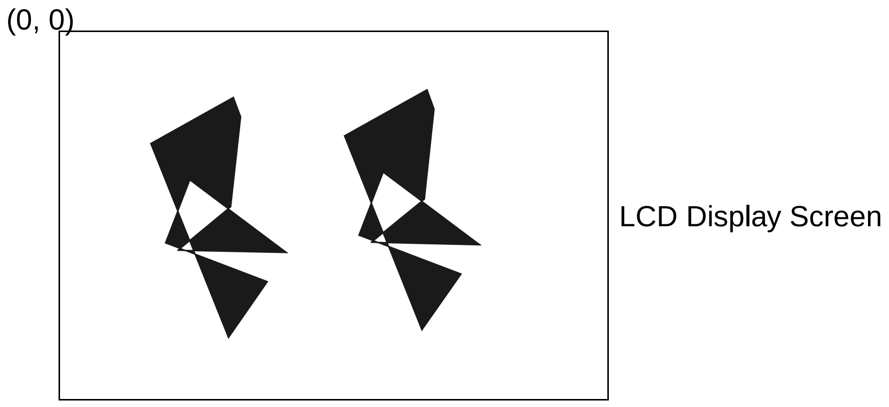


Figure 6-23 Screen output for *Example 6-24*

Example 6-25 LCD Display screen crosses the right boundary of the panning screen

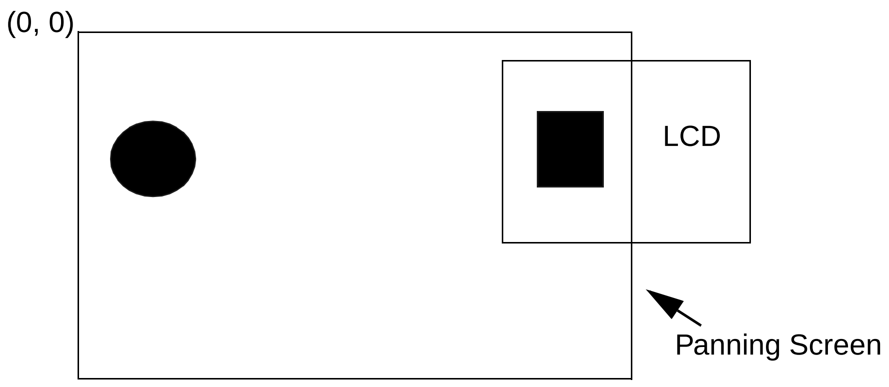


Figure 6-24 Screen output for *Example 6-25*

The following will be seen on LCD display screen:

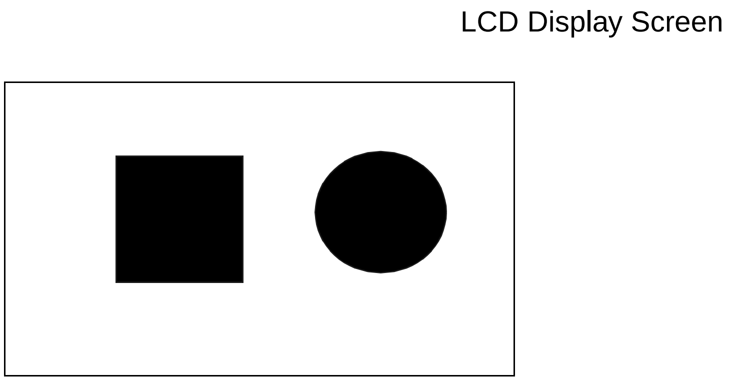


Figure 6-25 Screen output for *Example 6-25*

Example 6-26 When LCD Display screen crosses the bottom boundary of the panning screen

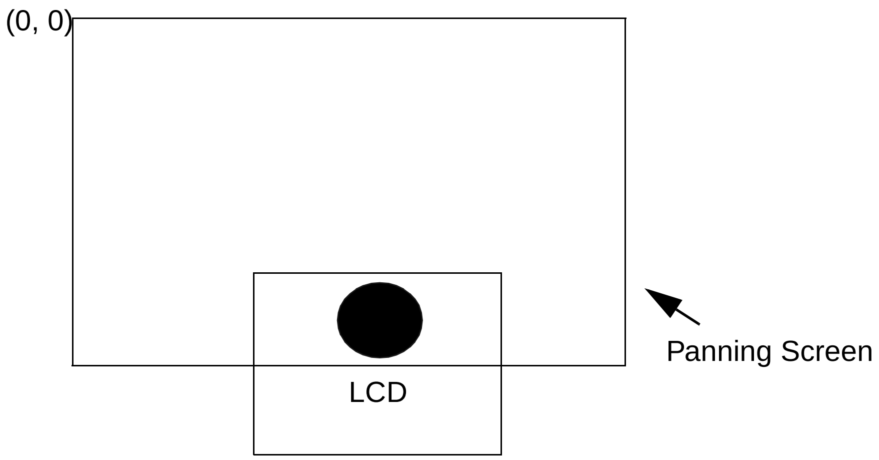


Figure 6-26 Screen output for *Example 6-26*

The following will be seen:

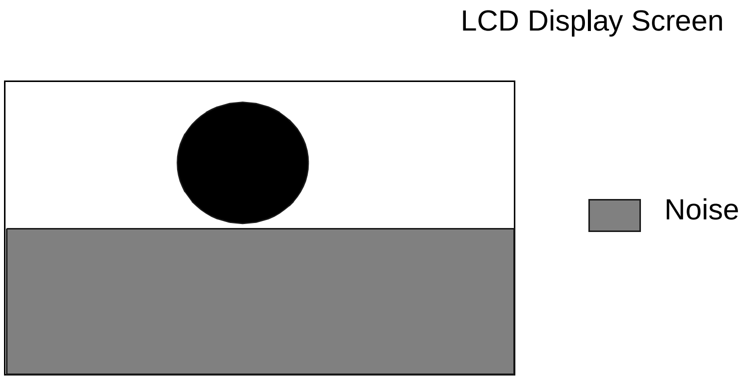


Figure 6-27 Screen output for *Example 6-26*

The pattern of the noise part of the display depends on the content of the memory that follows the panning screen. If the memory following the panning screen is all 0, the noise will appear as a blank image. If the memory following the panning memory is invalid, a bus address error will be generated.

6.27 Save a Rectangular Area from Panning Screen

STATUS **SaveRec**(P_U8 *bitmap*, U16 *xSrc*, U16 *ySrc*, U16 *width*, U16 *height*, U16 *reserved*)

This routine saves an image from the panning screen to memory.

Example 6-27 Save a bitmap

```
/* save the portion of image on panning screen from top left corner at (50, 50),
      width 320 and height 120 */
ret = SaveRec(bitmap, 50, 50, 320, 120, 0);
```

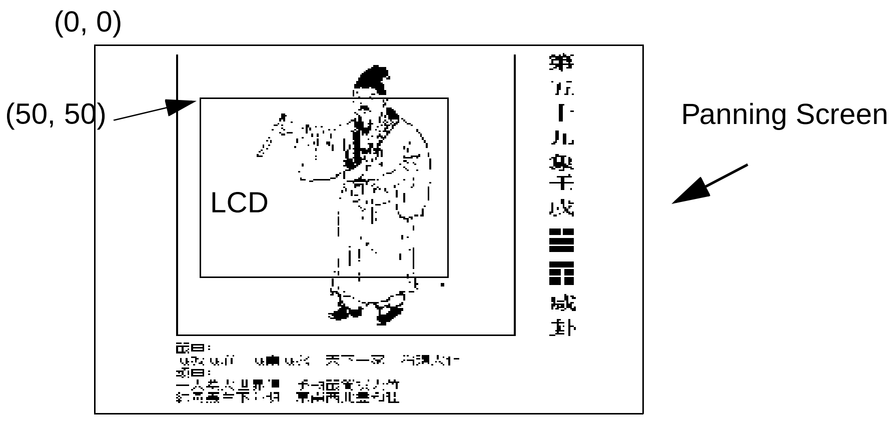


Figure 6-28 Screen output for *Example 6-27*

The calling of SaveRec(bitmap, 50, 50, 320, 120, 0) will save the top half of LCD display image into memory area pointed to by *bitmap*.

6.28 Exchange a Rectangular area with memory

STATUS **ExchangeRec**(U16 *xSrc*, U16 *ySrc*, U16 *width*, U16 *height*, P_U8 *reserved*)

This routine exchanges images between the panning screen and memory.

Example 6-28 Save a bitmap

```
/* exchange the image on panning screen with top left corner at (50, 50), width 320
      and height 120 to the image in memory pointed by bitmap */
ret = ExchangeRec(bitmap, 50, 50, 320, 120);
```

This example swaps the image pointed to by *bitmap* with the image in the rectangular region from top left corner at (50, 50) to bottom right corner at (369, 169). After this call, *bitmap* now points to the original image of the rectangular region (50, 50) to (369, 169), while the image pointed to by *bitmap* is now displayed on the rectangular region (50, 50) to (369, 169) on the panning screen.

6.29 Fill a Rectangular Area

STATUS **ClearRec**(U16 *greyLevel*, U16 *xSrc*, U16 *ySrc*, U16 *width*, U16 *height*, U16 *style*)

This routine fills an rectangular area with the specified grey level.

Example 6-29 Fill a rectangular region with BLACK and OR style

```
STATUS ret;

/* fill a rectangular area with top left corner at (300, 240), width 261 and height
      161 */
ret = ClearRec(BLACK, 300, 240, 261, 161, OR_STYLE);
```

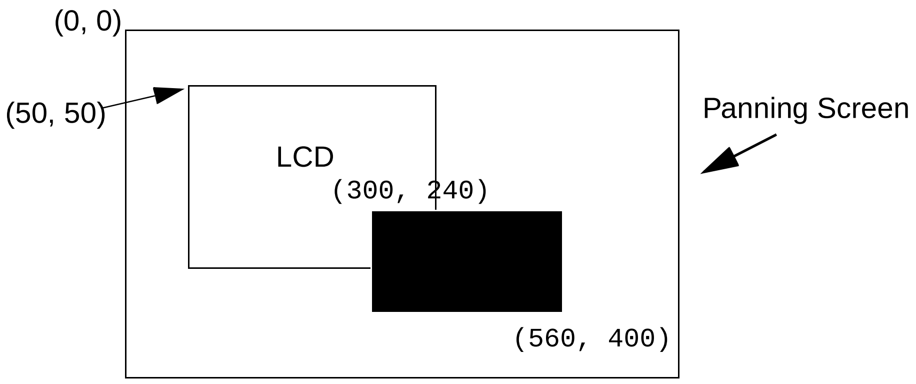


Figure 6-29 Screen output for *Example 6-29*

This example fills the rectangular region from top left corner at (300, 240) on panning screen with width 261 pixels and height 161 pixels.

6.30 Inverse a Rectangular Area

STATUS **InvRec**(U16 *xSrc*, U16 *ySrc*, U16 *width*, U16 *height*)

This routine will inverse the grey level of the rectangular area with top left corner at (xSrc, ySrc) and bottom right corner at (xSrc + width - 1, ySrc + height - 1).

Example 6-30 Inverse a rectangular region

```
STATUS ret;

/* inverse a rectangular area with top left corner at (250, 200), width 200 and
      height 100 */
ret = InvRec(250, 200, 200, 100);
```

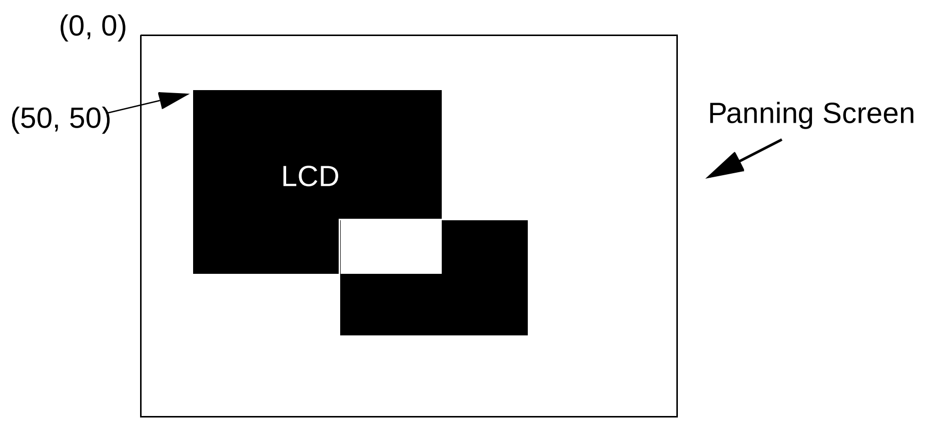


Figure 6-30 Screen output for *Example 6-30*

In this example, the LCD display screen is black and the rest of the panning screen is white. After inverting the rectangular region with top left corner at (250, 200) and 200 pixels wide by 100 pixels high, a white box can be seen in the LCD screen at bottom right position.

6.31 Hardware Cursor

When there is no hardware cursor in current task, the creation of hardware cursor requires to set the cursor characteristic and follows by calling CursorSetStatus(PPSM_CURSOR_ON).

When hardware cursor is created and it needs to be turned off, the application should call CursorOff(). If the application needs to turn on the cursor once again. It has to set the cursor characteristic and follows by calling CursorSetStatus(PPSM_CURSOR_ON) again.

When hardware cursor is to be suspended, the application should call CursorSetStatus(PPSM_CURSOR_OFF).

A application can change the hardware cursor to new position. It can inquire the hardware cursor status from the system. When the hardware cursor is ON, the calling of functions to change the size or position of the hardware cursor will have immediate effect.

6.31.1 Set Hardware Cursor Size

STATUS **CursorInit**(U16 *cursorWidth*, U16 *cursorHeight*)

This routine will change the hardware cursor width and height. The valid range for both width and height is from 1 through 31.

Example 6-31 When there is no hardware cursor in current task:

```
/* set hardware cursor position at (150, 150) */
CursorSetPos(150, 158);
```

```
/* set hardware cursor width to 15 and height to 15 pixels */
CursorInit(15, 15);

/* turn on the hardware cursor in full density mode */
CursorSetStatus(PPSM_CURSOR_ON);
```

The above will create a cursor at (150, 158) with 15 pixels wide by 15 pixels high, and will turn the cursor on.

6.31.2 Set Hardware Cursor Position

STATUS **CursorSetPos**(U16 *xPos*, U16 *yPos*)

This routine will set the hardware cursor top left corner position at (xPos, yPos).

Example 6-32 When the hardware cursor needs to be changed to other position:

```
/* set hardware cursor position to (15, 150) */
CursorSetPos(15, 150);
```

This will change cursor to new position at (15, 150)

6.31.3 Set Hardware Cursor Status

STATUS **CursorSetStatus**(U16 *status*)

This routine will change the hardware cursor status to one of the following states: PPSM_CURSOR_OFF, PPSM_CURSOR_ON or PPSM_CURSOR_REVERSED.

Table 6-12

Status name	Descriptions
PPSM_CURSOR_OFF	Temporarily turn cursor off
PPSM_CURSOR_ON	Turn cursor on
PPSM_CURSOR_REVERSED	Reverse cursor

Example 6-33 When hardware cursor is turned off after creation and it needs to be on with reverse video mode:

```
U16 x, y;

/* turn on hardware cursor in reverse video mode */
CursorSetStatus(PPSM_CURSOR_REVERSED);
```

6.31.4 Get Hardware Cursor Status

STATUS **CursorGetStatus**(P_U16 *status*)

This routine will return the current hardware cursor status. The status will be one of the following states: PPSM_CURSOR_OFF, PPSM_CURSOR_ON or

PPSM_CURSOR_REVERSED.

Example 6-34 When the hardware cursor status is needed:

```
U16 status;

/* get the hardware cursor status */
CursorGetStatus(&status);
```

6.31.5 Set Hardware Cursor Blinking Frequency

STATUS **CursorSetBlink**(U16 *frequency*)

This routine will set the hardware cursor blinking frequency to □requency?numb er of blinks per 10 seconds.

6.31.6 Turn Hardware Cursor Off

STATUS **CursorOff**(void)

This routine will turn off the hardware cursor permanently. In order to turn on the hardware cursor again, the application has to set the characteristics of the cursor and follows by calling CursorSetStatus(PPSM_CURSOR_ON).

6.32 Display Other Region of Panning Screen

STATUS **CursorSetOrigin**(U16 *xPos*, U16 *yPos*)

STATUS **LCDScreenMove**(U16 *x*, U16 *y*)

These two routines together will move the LCD display origin to (x, y) so that different regions of the panning screen can be displayed instantaneously. These two routines must be used together. In PPSM V3.1, these 2 routines are replaced by DisplayMove().

Example 6-35 Display other region of panning screen

```
U16 x=50, y=50;

/* set the LCD display screen origin to be (50, 50) on panning screen */
CursorSetOrigin(x, y);

/* change the hardware register to display the rectangular on panning screen with
      top left corner at (50, 50) */
LCDScreenMove(x, y);
```

The LCD Display screen will now display the rectangular region of the panning screen with top left corner at (50, 50).

6.33 Get LCD Display Origin on Panning Screen

STATUS **CursorGetOrigin**(P_U16 *xPos*, P_U16 *yPos*)

This routine will return the current LCD display screen origin relative to the panning screen.

Example 6-36 Get LCD display origin on panning screen

```
U16 x, y;

/* get the position of LCD display screen on panning screen */
CursorGetOrigin(&x, &y);
```

6.34 Allocate memory for Panning Screen

P_VOID **GetScreenMem**(U16 *width*, U16 *height*)

This routine will allocate memory for panning screen with specified size. However, the size of the new panning screen area cannot be larger than 64K. If no memory is available, it will return NULL.

Example 6-37 Allocate memory for panning screen

```
P_U8 screenPtr;

/* allocate memory for panning screen with width 320 and height 240 */
screenPtr = (P_U8)GetScreenMem(320, 240);
```

Chapter 7 Database Management

PPSM supports global database for applications to store and manage data. Data type may be formatted or unformatted.

The Database Tools enable applications to:

- create and delete multiple databases
- add and delete records from databases
- store, retrieve and modify data
- search for particular data in a database
- get status about a database

This chapter gives some example usages of the database tools to help user getting up to speed fast.

7.1 Data Format

PPSM database tools support two types of data, formatted and unformatted.

7.1.1 Formatted Data

Formatted data means text data, and is field accessible. Seven fields, as labeled in *Table 7-1*, will be used frequently in the PDA environment and have been predefined. Other Formatted Data fields can be defined by an application should need arise. When a new record is created, the number of Formatted Data fields is seven by default. PPSM allows users to add additional user-defined fields in a record. The number of maximum user defined field allowed in PPSM is defined by PPSM at compile time, currently it is five. Notice that searching is only done on Formatted Data, not Unformatted Data.

There is a size limit of 60 bytes imposed on each formatted data field.

Table 7-1 Predefined field format used in the PDA environment

Field Index	Field Name	Max. Field Size
DB_LAST	Last Name	60
DB_FIRST	First Name	60
DB_HOME	Home Phone	60
DB_OFFICE	Office Phone	60
DB_ADDRESS	Address	60
DB_FAX	Fax	60
DB_COMPANY	Company	60

Table 7-1 Predefined field format used in the PDA environment

Field Index	Field Name	Max. Field Size
1,2,3,4,5	5 additional fields	60

7.1.2 Unformatted Data

Unformatted data is data that has to be accessed as one complete block. There is only one unformatted data field allowed per record. The unformatted data can be one of the following eight types defined as follows:

type 0	text (ASCII code record)
type 1	decompressed PPSM LCD bitmap
type 2	compressed PPSM LCD bitmap
type 3	mixed text and graphics
type 4	reserved
type 5	text followed by decompressed PPSM LCD bitmap
type 6	text followed by compressed PPSM LCD bitmap
type 7	text followed by mixed text and graphics

PPSM will keep track of the size of the unformatted data field of a record in the record information field.

7.2 The Database Manipulation Tools

PPSM provides a set of tools to operate on databases and records. Conceptually, they can be grouped together in the following way:

1) Operate at the database level:
   - Add or delete a database.
   - Inquire the total number of database present in the current environment.
   - Set or clear the database secret flag.
   - Interrogate the database secret flag.
2) Operate at the individual record level:
   - Add or delete a record.
   - Add a blank record to the top of the record list.
   - Append a blank record after a specified record in the record list.
   - Write or read formatted data of a record.
   - Write or read unformatted data of a record.
   - Get the ID of the first record in the list.
   - Get the ID of the previous or next record in the list.
   - Search for a record using a formatted data field as a key.
   - Set or clear the record secret flag.
   - Interrogate the record secret flag.
   - Inquire the total number of record present in a particular database.

7.3 Creating and Editing a Database

Whenever a record is created, PPSM will pass back a unique record identifier. Application will need to use this identifier as the key for subsequent access of that particular record.

The following example illustrates a routine build database and its associated memory variables which create a database, add new record and initialize four formatted data field in a record.

Note that user must have already declared the variable gAddBkDBaseID before calling DBAdd(). It is important that the database ID must remain intact since access to the database need this as the key.

Note also that in the example, recID is used as a temporary variable only, the creation of the next record will overwrite the former recID. This is permissible because of the linked list structure of the record list, we can traverse the list to retrieve the record that we need without needing you keep track of every record ID. However, for those frequently called record, it would be better to allocate a memory variable to keep its record ID for speedy access.

Example 7-1 Create a database, add record, write data to record, and read number of records in database

```
/* Database identifier *
U32 gAddBkDBaseID;
      /* Data to be written in RECORD 1 */
TEXT grec1LName[5]={'A','d','a','m',0};
TEXT grec1FName[4]={'K','e','n',0};
TEXT grec1Phone[9]={'2','2','2','-','6','7','8','9',0};
TEXT grec1Add[17]={'1','0',' ','M','a','v','e','n',' ','A','v','e',0};

      /* Data to be written in RECORD 2 */
TEXT grec2LName[8]={'A','p','p','l','e',0};
TEXT grec2FName[4]={'J','o','e',0};
TEXT grec2Phone[9]={'6','6','6','-','1','3','3','1',0};
TEXT grec2Add[17]={'1','2',' ','M','a','i','n',' ','S','t','r','e','e','t',0};

    STATUS buildDBase(void)
    {
      STATUS ret;
      U32 recID;
      S32 numRec;

      ret = DBAdd(&gAddBkDBaseID);
      if(ret != PPSM_OK) return(-1);
            /* Add Record 1 */
       DBAddRecord(gAddBkDBaseID,&recID, 0);
       DBChangeStdData(gAddBkDBaseID,recID, DB_LAST,grec1LName);
       DBChangeStdData(gAddBkDBaseID,recID, DB_FIRST,grec1FName);
       DBChangeStdData(gAddBkDBaseID,recID, DB_HOME,grec1Phone);
       DBChangeStdData(gAddBkDBaseID,recID, DB_ADDRESS,grec1Add);

            /* Add Record 2 */
       DBAddRecord(gAddBkDBaseID,&recID, 0);
       DBChangeStdData(gAddBkDBaseID,recID, DB_LAST,grec2LName);
       DBChangeStdData(gAddBkDBaseID,recID, DB_FIRST,grec2FName);
       DBChangeStdData(gAddBkDBaseID,recID, DB_HOME,grec2Phone);
       DBChangeStdData(gAddBkDBaseID,recID, DB_ADDRESS,grec2Add);
           /* Read back no of record in the database, should be 2*/
       DBReadTotalNumberRecords(gAddBkDBaseID, &numRec);
      }
```

7.4 Searching and Retrieving Data

The following example uses the database created from *Example 7-1*, and searches for a particular record using a formatted data field as the key. It is good programming practice to check the returned status of the call DBSearchData() ensuring that a match is found before using the recID returned for subsequent operation.

Example 7-2 Search a database record list using a formatted data field and retrieve record data

```
/* Global database identifier */
U32 gAddBkDBaseID;
/* This is the search key */
TEXT grec1FName[4]={’K’,’e’,’n’,0};
    STATUS searchRec(void)
    {
      STATUS ret;
      U32 recID;
      /* Pointers to the formatted data field to be read out */
      P_TEXT tempLname, tempFname, tempPhone, tempAddress;

      /* Search for a record in the record list with the DB_FIRST
         formatted data field matching the key string grec1FName */
      ret = DBSearchData(gAddBkDBaseID,DB_FIRST,grec1FName,&recID);

      if (ret != PPSM_OK) return (-1);    /* Search unsuccessful! */
      /* recID now is the record which match the search key           */
      /* We can now access the data contained in the record via recID */
      DBReadData(gAddBkDBaseID, recID, DB_LAST, &tempLname);
      DBReadData(gAddBkDBaseID, recID, DB_FIRST, &tempFname);
      DBReadData(gAddBkDBaseID, recID, DB_HOME, &tempPhone);
      DBReadData(gAddBkDBaseID, recID, DB_ADDRESS, &tempAddress);
      /* tempLname, tempFname, tempPhone, tempAddress now points to
         data field stored in the record identified by recID   */
       } /* end searchRec() */
```

7.5 Navigating along a Record List

The following example uses the record list navigation tools to access record sequentially. It is assumed that a database with the identifier gAddBkDBaseID has already existed. Initially, the first record ID in the record list is retrieved. Then a while loop is set up to access record in the list sequentially. Termination of the loop is by using the botFlag passed to the routine DBGetNextRecID(). When this flag is set to 1, it signifies that the current record is the last record of the record list.

Example 7-3 Use the record list navigation tools to access record sequentially

```
/* Global database identifier */
U32 gAddBkDBaseID;

 STATUS seqAccessofRec(void)
 {
    U32 RecId,nextRecId;
    S32 srchToken = 1;
    U16 botFlag = 0;
    P_TEXT stdDataOut;
    STATUS ret;

    /* Get the first record ID for the database */
    ret = DBGetFirstRecID(gAddBkDBaseID, &RecId);
```

```
   /* Check if the database record list is empty */
   if (ret == PPSM_ERROR) return(-1);

   /* Read back data */
   DBReadData(gAddBkDBaseID, RecId, DB_FIRST, &stdDataOut);
   .
   /* Do something here */
   .
   /* Get next record */

   while(srchToken)
     {
     DBGetNextRecID(gAddBkDBaseID, RecId, &nextRecId, &botFlag);
      /* The RecId passed is already at the end of list, exit loop */
     if(botFlag == 1){ srchToken = 0; break;}
     /* Read back data */
     DBReadData(gAddBkDBaseID, nextRecId, DB_FIRST, &stdDataOut);
     .
      /* Do something here */
      .
      /* Continue search   */
      RecId = nextRecId;
      }   /* end while (srchToken) */
}    /* end of seqAccessofRec() */
```

Chapter 8 Text Display Management

Applications must map the text with its properties described in a text template to an area on the display screen, called the text display area, before any text can be seen. This chapter describes the set of text tools provided by PPSM to manage the display of text on the panning display screen.

PPSM supports 16-bit text data representation which allows the support of any coded languages. The default is the support for various font types and sizes of Asian and English characters display. The low level font driver supports both the scalable and bitmap font technologies. PPSM provides a set of default English fonts with size 8 x 10 and 16 x 20. If other fonts are needed, please contact the ISV.

8.1 Text Representation

Both ASCII and Asian characters are stored internally as 16-bit values (type TEXT) system-wide. For 8-bit ASCII, they are zero-extended to 16-bit. For Chinese, both GB and BIG-5 code formats are supported.

The most significant byte of a 16-bit word is used to distinguish between 16-bit zero-extended ASCII and Asian codes; all 16-bit ASCII characters have a zero most significant byte while all Asian codes will have a non-zero most significant byte.

8.2 Text Display Area

Text can be displayed anywhere within the panning screen (refer to *Section 2.3.2.2 - Panning Screen*). Text is displayed starting at a specified location in a row by column format one character at a time as shown in *Figure 8-1*.

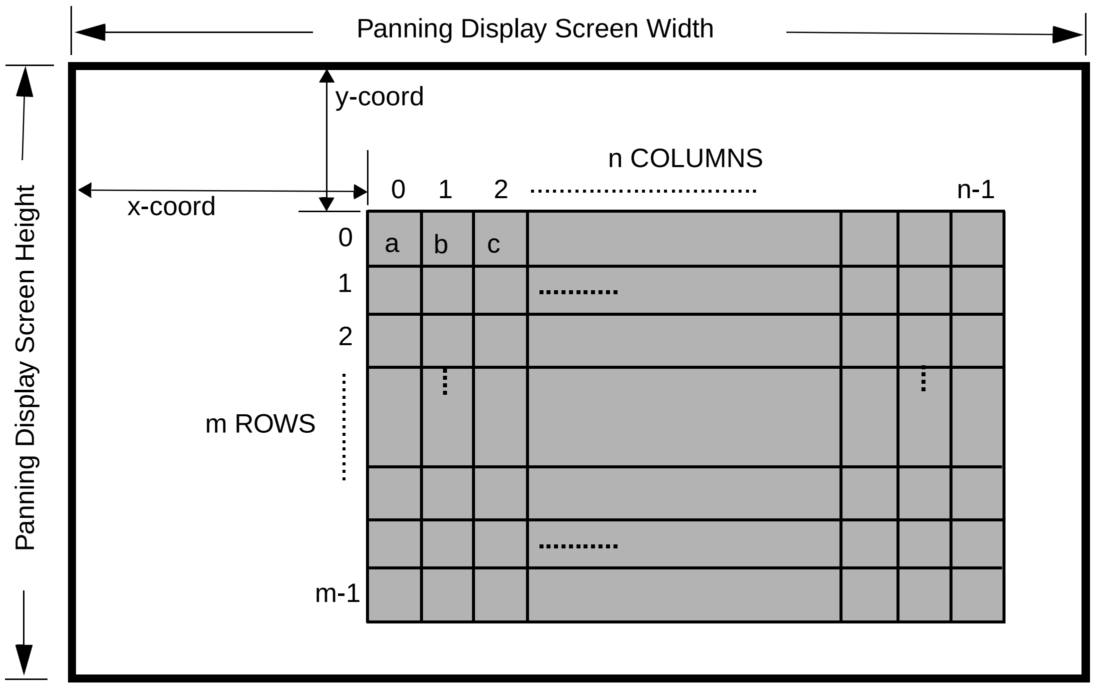


Figure 8-1 A Text Display Area on the Panning Display Screen

8.3 Text Templates

A text template refers to a collection of text properties that describes the text to be displayed. These text properties include font type, font size, grey level, output style, coordinates and size of the text display, and the position of the display soft cursor. These text templates are independent of the text itself and provide the flexibility for applications to change the appearance of text in a collective and efficient manner. Applications can create and delete the text templates at their discretion on an as needed basis. The soft cursor in text is an invisible position indicator showing where the text should be mapped.

8.3.1 Creating text templates

STATUS **TextCreate**(P_U32 *templateId*)

A text template needs to be created before any text can be displayed. A unique unsigned 32-bit text template identifier is returned from the system for each text template created. This text template identifier is used for future references to the created text template. Refer to *Table 8-4* for the default values of the text properties when a text template is created.

Example 8-1 Create a text template

```
U32 tId;  /* textId for the text template */

if(TextCreate(&tId) != PPSM_OK)
```

```
    return PPSM_ERROR;

```

8.3.2 Deleting text templates

STATUS **TextDelete**(P_U32 *templateId*)

When a text template is not needed anymore, applications should delete it to free up space that is being used to store the text properties. The text template identifier returned by TextCreate() is used to specify which text template to be deleted.

Example 8-2 Delete a text template

```
/* Delete the text when it’s no longer needed */
if(TextDelete(tId) != PPSM_OK)
  return PPSM_ERROR;
```

8.4 Text Properties

Text properties describes the layout and appearance of the text to be displayed on the panning display screen. These text properties, stored collectively in a text template, include the position and size of the text display area, the outlook of the text characters, the font attributes, and the position of the text soft cursor within the text display area. Refer to *Table 8-4* for the default values of these properties of a text template when it is created.

8.4.1 Setting Text Display Layout

STATUS **TextSetDisplay**(U32 *templateId*, U16 *xPos,* U16 *yPos,* U16 *width*,
U16 *height*, U16 *cursor*)

The text display layout of a text template is rectangular and must reside within the boundary of the panning screen. The layout is anchored by the xy-coordinate of the upper left corner, and the width and height of the area in number of characters. The size of the text display area in number of pixels will vary according to the size of the selected font type. The specified cursor positions must lie within the range of 0 and one less than the number of characters the text display area can display.

In *Figure 8-1*, the text display area is located at location (x, y) and it is m rows by n columns in size. This text display area can be moved around as the application wishes.

8.4.2 Setting Text Outlook

STATUS **TextSetOutlook**(U32 *templateId*, U16 *outputStyle*, U16 *greyLevel*)

The text outlook of a text template are the output style and the grey level of the text to be displayed on the text display area. The new output style and grey level will take effect on subsequent text mapped using the text template after it□ been modified.

8.4.2.1 Text output style

The output style defines an operation between the text bitmap and the existing image at a specified display location. Five output styles are supported. The text bitmap can replace, OR with, AND with, exclusive OR with, or be inverted to replace the existing image.

Table 8-1 Supported Output Styles

Output Styles	Operation
REPLACE_STYLE	Replace
OR_STYLE	Or with
AND_STYLE	And with
EXOR_STYLE	Exclusive-Or with
INVERSE_STYLE	Invert and replace

8.4.2.2 Text Grey Levels

Depending on the hardware system, up to four grey levels (0 to 3) can be supported. For details about the grey levels, refer to *Section 6.4 - 1 bit-per-pixel Graphics* and *Section 6.5 - 2 bits-per-pixel Graphics*.

Table 8-2 Supported Grey Levels

Grey Level Values	Color
BLACK	Black
DARK_GREY	Dark Grey
LIGHT_GREY	Light Grey
WHITE	White

8.4.3 Setting Font Attributes

STATUS **TextSetFont**(U32 *templateId*, P_FONTATTR *pFontAttr*)

Font attributes includes font type, font size (where applicable), and any attributes (e.g. outline) that□ related to a particular font type.

A font attribute data structure, called FONTATTR, needs to be filled with the desired values in order for these font attributes to be modified. These new font attributes will take effect when text is displayed using the text template which has these new values set. The following is the font attribute structure defined in ppsm.h:

```
typedef struct
{
      U16         type;             /* font type */
      U16         width;            /* font width (in #pixels) */
      U16         height;           /* font height (in #pixels) */
      U16         attrib;           /* other font attributes */
```

```
        P_U8        bitmap;           /* pointer to character bitmap */

    } FONTATTR, *P_FONTATTR;
```

8.4.3.1 Font Types

Eight font types are supported. Small Normal and Small Italic are 8 x 10 pixels English fonts. Large Normal and Large Italic are 16 x 20 pixels English fonts. GB Normal is 16 x 16 Chinese font in GB code format. Chinese Normal is the same as GB Normal (for backward compatibility). BIG5 Normal is 16 x 16 Chinese font in BIG5 code format. BIG5 Variable is a variable size font in BIG5 code format.

Table 8-3 Supported Font Types and Sizes

Output Styles	Operation
SMALL_NORMAL_FONT	8 x 10 English Normal
SMALL_ITALIC_FONT	8 x 10 English Italic
LARGE_NORMAL_FONT	16 x 20 English Normal
LARGE_ITALIC_FONT	16 x 20 English Italic
GB_NORMAL_FONT	16 x 16 GB Normal
CHINESE_NORMAL_FONT	same as GB_NORMAL_FONT
BIG5_NORMAL_FONT	16 x 16 BIG5 Normal
BIG5_VARIABLE_FONT	Variable size BIG5

Note: Asian fonts are supplied by third parties.

8.4.3.2 Font Sizes

If a scalable font engine is available, the BIG5_VARIABLE_FONT font type allows an application to specify the font width and height of the characters in a text template. The minimum and maximum width and height supported depend on the particular scalable font engine being provided.

All the other font types are fixed size fonts. Any attempt to modify the font width and height of those font types is ignored.

8.4.3.3 Special font attributes

No other special font attributes are currently being supported. So, attempts to modify the font attributes field of a text template is ignored. The attribute field is reserved for future extensions and is set to zero.

Table 8-4 Text Properties Default Values

Text Properties	Default Value
(x,y)-coordinate of the origin (top left corner) of the text display area	(0, 0)
Width of text display area in number of characters	0
Height of text display area in number of characters	0
Character cursor position relative to origin of text display area	0
Font type	SMALL_NORMAL_FONT
Font width	8
Font height	10
Special font attributes	0
Text grey level value	BLACK
Text output style	OR_STYLE

Example 8-3 Setting text properties

```
U32    tId;               /* text template id */
                          /* text to be displayed */
TEXT   moto[] = {’M’, ’o’, ’t’, ’o’, ’r’, ’o’, ’l’, ’a’, 0};
                          /* this is to initialize every ASCII character in
                             2-byte format with high byte being zero */
U16    len;               /* # chars to be displayed */
FONTATTR     fontAttr;    /* font attributes */
.
.
.
/* create a text template */
TextCreate(&tId);

/* calculate # chars to be displayed */
len = Strlen(moto);

/* subsequent text displayed with text template tId will be located at (102, 0),
      length of moto characters wide and 6 characters high, starting at text cursor
      position zero. */
TextSetDisplay(tId, 102, 0, len, 6, 0);

/* subsequent text displayed with text template tId will be exclusive OR□d with
      the existing image on screen, with grey level BLACK. */
TextSetOutlook(tId, EXOR_STYLE, BLACK);

/* set up font attributes */
fontAttr.type = BIG5_VARIABLE_FONT;
fontAttr.width = fontAttr.height = 20;
fontAttr.attrib = 0;

/* subsequent text displayed with text template tId will be of BIG5 Variable font
      type of size 20x20 pixels. */
TextSetFont(tId, &fontAttr);

/* delete unused text template */
```

```
TextDelete(tId);
.
.
.
```

8.5 Text Mapping

Mapping functions are provided for applications to display and remove text on the panning display screen area. The display and removal of text are tied to a text template.

8.5.1 Displaying text

STATUS **TextMap**(U32 *templateId*, P_TEXT *buffer*, U16 *numChar*)

The given text is displayed, one character at a time, starting at the current character cursor position of the text display area and with text attributes as described by the specified text template.

There is no word-wrap function. Text is treated as individual characters, i.e. characters of a word that extends beyond a row will appear on the next row of the text display area. Text displaying stops when the character cursor position is at the end of the text display area, when all characters supplied by the application are mapped, or when *numChar* characters are mapped. After displaying characters on panning screen, character cursor position will be advanced to the next available position, or (the end of the template + 1) if the last character displayed is at the end of the template. Any out standing characters are going to be ignored without returning any error.

Example 8-4 Display text on text display area

```
STATUS  DisplayString(U8 font, U8 style, U8 greylev, U16 xSrc, U16 ySrc, TEXT str[])
{
  U16 len;  /* length of the string */
  U32 tId;  /* textId for the text template */
.
.
.
  /* get the string length */
  if (len = Strlen(str))
    {
      /* Set up the text attributes */
      if(TextSetup(tId, font, style, greylev, xSrc, ySrc, len, 1) != PPSM_OK)
       return PPSM_ERROR;

      /* Map the text to the screen */
      if(TextMap(tId, (P_TEXT)str, len) != PPSM_OK);
      return PPSM_ERROR;
```

8.5.2 Removing text

STATUS **TextUnmap**(U32 *templateId*)

The unmapping of text means clearing the entire text display area. The location and size of the text display area to be cleared is specified in the given text template.

8.6 Text character cursor position

The character cursor position determines where within the text display area text will be displayed next. This position is relative to the origin of the text display area specified in the given text template. The range of valid cursor positions is zero through one less than the size of the text display area in number of characters.

In *Figure 8-1*, the range of valid cursor positions is zero through (m * n - 1), and the current cursor position is 3 after □bc?is displayed

8.6.1 Setting the character cursor position

STATUS **TextSetCursor**(U32 *templateId*, U16 *cursor)*

Setting the character cursor position of the text display area of the specified text template to the given value. Subsequent displaying of text start at this new character cursor position.

Example 8-5 Set character cursor position

```
static U32 gTextId, gTmpTextId;
.
.
.
         /* Clear the text on the display and reset cursor */
              TextUnmap(gTextId);
              TextSetCursor(gTextId, 0);
```

8.6.2 Reading the character cursor position

STATUS **TextReadCursor**(U32 *templateId*, P_U16 *cursor*)

Applications can inquire the current character cursor position of a text display area specified by a text template. The returned character cursor position is where text will be displayed next.

Example 8-6 Set and read the character cursor position

```
U32    tId;         /* text template id */
TEXT   moto[] = {’M’, ’o’, ’t’, ’o’, ’r’, ’o’, ’l’, ’a’, 0};/* text to be displayed
       */
U16    len;   /* # chars to be displayed */
U16    curPos;/* cursor position */
.
.
.
/* create a text template */
TextCreate(&tId);

/* calculate # chars to be displayed */
len = Strlen(moto);

/* set up text properties. */
TextSetup(tId, LARGE_NORMAL_FONT, EXOR_STYLE, 3, 102, 0, len, 6);

/* set current character cursor position to beginning of 2nd row in the text template
       */
TextSetCursor(tId, len);

/* display □otorola?using the modified text properties */
```

```
TextMap(tId, (P_TEXT)moto, len);

/* read current character cursor position (should be at beginning of 3rd row in this
      case) */
TextReadCursor(tId, &curPos);
.
.
.
```

Chapter 9 Timer Management

PPSM uses one DragonBall^{TM} family timer to maintain a continuous 32-bit system reference timer and the system clock. The reference timer has a resolution of 100 microseconds and the system clock has a resolution of 1 second. A set of timer tools are included for applications to set the system clock, system date, periodic alarm, clock alarm and time-out.

PPSM manages all the timer devices. The tools allow the application to set a specific time-out or alarm, then continue with other operations. When the appropriate alarm time or time-out period is reached, PPSM generates the soft interrupt to notify the application of the event.

The system default time at power reset is at 9:00am, 1st of January, 1997.

Two types of interrupt messages are defined for timer functions:

` IRPT_RTC
` IRPT_TIMER

IRPT_RTC message is sent to the application for clock alarm and periodic alarm.

IRPT_TIMER message is sent to the application for time-out.

If the timeout or alarm happens when the current task is not the timer nor the alarm task, the timer or alarm task will be swapped in once all messages in current task are handled.

9.1 Reading System Date and Time

STATUS **DateTimeRead**(P_U16 *year*, P_U16 *month*, P_U16 *day*, P_U16 *hour*, P_U16 *minute*, P_U16 *second*)

Read the system date and time.

9.2 Setting System Date and Time

STATUS **DateTimeSet**(U16 *year*, U16 *month*, U16 *day*, U16 *hour*, U16 *minute*, U16 *second*)

Set the system date and time. The *year* cannot be set less than 1900.

9.3 Reading Clock Alarm

STATUS **AlarmRead**(P_U16 *year*, P_U16 *month*, P_U16 *day*, P_U16 *hour*, P_U16 *minute*, P_U16 *second*)

STATUS **AlarmReadId**(U32 *alarmId*, P_U16 *year*, P_U16 *month*, P_U16 *day*, P_U16 *hour*, P_U16 *minute*, P_U16 *second*)

Read the coming clock alarm time using AlarmRead() and read the specific clock alarm time by using AlarmReadId(). If no alarm is set, all arguments will return zero.

9.4 Setting Clock Alarm

STATUS **AlarmSet**(U16 *year*, U16 *month,* U16 *day*, U16 *hour*, U16 *minute,* U16 *second*)

STATUS **AlarmSetId**(P_U32 *alarmId*, U16 *year*, U16 *month,* U16 *day*, U16 *hour*, U16 *minute,* U16 *second*)

Set the clock alarm. When the alarm time is reached, PPSM will generate a soft interrupt to the application that called this tool. The interrupt message type will be IRPT_RTC. If AlarmSetId() is used, the alarm id. will be returned with the IRPT_RTC when time is reached.

9.5 Clearing Clock Alarm

void **AlarmClear**(void)

void **AlarmClearId**(U32 *alarmId*)

Clear the clock alarm in current task using AlarmClear() or clear specific alarm using AlarmClearId(). PPSM will no longer generate the IRPT_RTC message to the application.

9.6 Setting Periodic Alarm

STATUS **SetPeriod**(U16 *period*)

STATUS **SetPeriodId**(P_U32 *alarmId*, U16 *period*)

Set or clear the periodic alarm. PPSM generates periodic interrupts to the application that calls this tool. The HOUR periodic interrupt is applicable to MC68EZ328 only. If SetPeriodId() is used to set a periodic alarm, the alarm id. will be returned. The period that is allowed are:

RTC_PERI_NONE	Disable periodic interrupt
RTC_PERI_SECOND	Second periodic interrupt
RTC_PERI_MINUTE	Minute periodic interrupt
RTC_PERI_HOUR	Hour periodic interrupt
RTC_PERI_MIDNIGHT	Midnight periodic interrupt

RTC_PERI_NO_SECOND	Disable second interrupt for current task
RTC_PERI_NO_MINUTE	Disable minute interrupt for current task
RTC_PERI_NO_HOUR	Disable hour interrupt for current task
RTC_PERI_NO_MIDNIGHT	Disable midnight interrupt for current task

9.7 Setting Timeout

STATUS **Timeout**(U32 *millisecond*)

STATUS **TimeoutId**(P_U32 timerId, U32 *millisecond*)

General time-out tool. PPSM will generate a single soft interrupt message to the caller once the time-out period has elapsed. An input value of zero will immediately disable the time-out function for the current task including those timeout set by reference timer alarm tools. TimeoutId() will output the *timerId* which will be returned in IrptGetData() if time elapsed.

IRPT_TIMER is the interrupt message sent to the caller application when time-out occurs.

9.8 Setting Input Timeout

STATUS **InputTimeout**(U32 *millisecond*)

Set the repetitive time-out period for data input from the touch panel. This time-out routine is an explicit time-out routine dedicated for the pen input device.

Once this time-out is activated, the specified time-out period count down begins immediately after a valid pen input stroke. If the time-out period expires before the next pen input occurs, PPSM will generate a timer interrupt to the caller; otherwise, the time-out period is reset for the next pen input. This is a repetitive time-out because once activated, the timer will continuously be set for time-out after each pen input stoke until the time-out is cancelled.

IRPT_TIMER is the interrupt message sent to the caller application. The timer id. returned in IrptGetData() will be 0xFFFFFFFF to distinguish it from the normal timer set by Timeout(), TimeoutId(), RefTimeAlarm(), etc.

To cancel the input time-out for current task, call this function with zero as the argument.

9.9 Continuous Reference Timer

PPSM provides a continuous 32-bit reference timer to applications. This 32-bit value wraps around about every 5 days, but PPSM takes care of the wrap-around condition, making it transparent to the application. Applications can select to use either a resolution of 1 millisecond unit or 100 microsecond unit. The following tools allow the user to make use of this reference timer for functions, such as time-stamping and time-out.

Note that there are two sets of timer tools, one for millisecond resolution, one for 100 microsecond resolution. The reference value returned by these two separate sets of tools should NOT be mixed. That is, values returned from the millisecond tools cannot be used in the 100 microsecond tools.

The millisecond resolution timer tools are named with prefix "RefTime", and the 100 microsecond resolution tools are named with prefix "RefFineTime".

9.10 Read The Reference Timer

U32 **RefTimeRead**(void)

U32 **RefFineTimeRead**(void)

Read the reference timer value. The return value is an unsigned 32-bit integer representing the current reference timer value, either in millisecond resolution for RefTimeRead(), or in 100 microsecond resolution for RefFineTimeRead().

9.11 Set Reference Timer Alarm

STATUS **RefTimeAlarm**(U32 *alarmTime*)

STATUS **RefFineTimeAlarm**(U32 *alarmTime*)

STATUS **RefTimeAlarmId**(P_U32 *alarmId*, U32 *alarmTime*)

STATUS **RefFineTimeAlarmId**(P_U32 *alarmId*, U32 *alarmTime*)

Set the alarm time, using the reference value as reference. This is a relative alarm tool. When using this tool, the input argument is the time that the system will generate an alarm interrupt to the caller application. This value can easily be obtained by calling the respective RefTimeRead() or RefFineTimeRead() tool. The *alarmId* output from this functions will be returned in IrptGetData() once the alarm time is reached.

9.12 Compute Reference Times Differences

U32 **RefTimeDiff**(U32 *beginTime,* U32 *endTime*)

U32 **RefFineTimeDiff**(U32 *beginTime,* U32 *endTime*)

Compute the difference in time for the given two reference times. The return value is in millisecond resolution for RefTimeDiff(), and in 100 microsecond for RefFineTimeDiff().

Example 9-1 Timer Usage

```
U32     (*TimeRead)();

STATUS  (*TimeAlarm)(U32 alarmTime);
U32     (*TimeDiff)(U32 beginTime, U32 endTime);
.
.
STATUS  RefTimer(void)
{
    .
    .
    while ( 1 )
    {

          /* Initialize all the variable for holding the time value */

          gOldTime    = 0;
          gNewTime    = 0;
          gDiffTime   = 0;

          /* Set up time function for reference timer to read the time,
           * to start the alarm and to calculate the time difference.
           */
          if (gUnit == MILLI_SECOND)     /* Unit in millisecond */
           {
             /*  Assign the function pointer to its corresponding reference
              *  timer function.
              */
             TimeRead  = RefTimeRead;
             TimeAlarm = RefTimeAlarm;
             TimeDiff  = RefTimeDiff;
           }
          else if (gUnit==MICRO_SECOND) /* Unit in microsecond */
           {
             TimeRead  = RefFineTimeRead;
             TimeAlarm = RefFineTimeAlarm;
             TimeDiff  = RefFineTimeDiff;
           }
           SetUnit(MILLI_SECOND); /* use RefTime toolset */
           .
           if (*inData == PPSM_ICON_PEN_UP)
           {
                 .
                 .
                 if (id == readTimerId)
                 {
                       /* Put the previous time to the gOldTime buffer, then read
                        * the latest time and store in the gNewTime. Get the
                        * difference of both and store in the gDiffTime.
                        */

                       gOldTime = gNewTime;
                       gNewTime = (*TimeRead)();
                       gDiffTime= (*TimeDiff)(gOldTime, gNewTime);
                       DisplayTime();
                 }/*if readTimerId*/
                 .
                 .
           }
    }/*while*/
}/*RefTimer*/
```

Chapter 10 Memory Management

In order for PPSM to manage system memory usage, the standard memory management tools provided by the compiler are disabled. PPSM provides a set of its own memory tools that allow the application programmers to dynamically allocate memory from the system. The size of this dynamic memory available to PPSM is specified in the Linker Specification File (Refer to *Chapter 34 - Linker Specification File*).

Note: For allocating memory to panning screen, a special memory allocation function called GetScreenMem() in Chapter 6 - Using Graphics Tools is used.

10.1 Allocating Memory

void ***Lmalloc**(U32 size)

void ***Lcalloc**(U32 size)

Memory can be allocated to the application at run time. PPSM returns to the caller a pointer to a block of available memory of the specified size. The memory returned to the caller is not initialized if Lmalloc() is called, or is initialized to zero if Lcalloc() is used. No automatic boundary checking is performed on the memory when used by the caller. The size of the largest block of memory can be allocated through Lmalloc() can be found by calling Lmalloc(LARGEST_MALLOC_SIZE).

If no memory is left in the system, these routines return a NULL.

The actual size of memory allocated by the system is larger than the size requested by user. A header is embedded in the allocated memory block for memory management. Nevertheless, it is transparent to user. User can directly use the required size of memory block start at the returned address if the returned value is not NULL.

10.2 Freeing Memory

void **Lfree**(void *ptr)

When an application finishes with a block of dynamically allocated memory, the memory can be recycled by using the Lfree() tool. It puts the memory block back into the system heap and the memory is ready for allocation again.

The pointer passed into this routine must be a valid pointer returned from Lmalloc() ,Lcalloc() or Lrealloc().

10.3 Reallocating Memory

void ***Lrealloc**(void *src, U32 size)

Moving of memory. This routine re-allocates the memory that is being used in the system from one location to another. It allocates a new area, then copies the content from the old location to the new area and free up the old memory, putting it back into the system heap. The purpose of this routine is for defragmentation of the system memory.

10.4 Copying Memory

STATUS **MoveBlock**(P_U32 *srcAddr*, P_U32 *destAddr*, U32 *size*)

Copying memory from one region to another. This tool can cope with over-lapping area. It performs memory copy in 32-bit operations whenever possible.

10.5 Inquiring Memory

STATUS **TaskMemUsed**(U32 *taskId,* P_U32 *pSizeUsed*)

U32 **TotalMemUsed**(void)

U32 **TotalMemSize**(void)

S32 **TaskStackAvail**(void)

Memory allocated to the application and the whole system can be inquired at run time. PPSM returns to the caller the total number of bytes of memory allocated to the task with the given task identifier when calling TaskMemUsed(), or number of bytes of memory allocated to the whole system when calling TotalMemUsed(). PPSM returns the number of bytes of memory on the system can be allocated through Lmalloc(), Lcalloc() or Lrealloc() when calling TotalMemSize().

PPSM returns to the caller the total number of bytes of stack can still be used by current task when calling TaskStackAvail(). Positive returned value indicates stack has not been used up, negative value implies stack has already overflowed.

User can inquire the size of the largest continuous memory block by calling Lmalloc() with input flag LARGEST_MALLOC_SIZE.

Chapter 11 Power Management

PPSM utilizes the power control module of DragonBall^{TM} to implement a set of power management tools to achieve system power saving.

The Power Management Tools enable applications to:

- switch to one of the power saving modes
- control the duty cycle of the processor for each application in Normal mode
- switch automatically to a lower power saving mode when system is idle
- control user defined I/O ports in any of the power saving mode transition

Applications can choose to:

- control the system□ power management features directly, or
- use the PPSM□ automatic power management features.

By default, the system will go to doze mode if there is no more task swapping nor message waiting to be served in current task. So the default state is 0 sec. doze period and sleep period counting will start if it□ set in SetSleepPeriod().

11.1 Power Control Module

PPSM makes use of the Power Control Module, PCM, to improve system power efficiency. It allows the allocation of system clock cycles to the CPU core under software control. System clocks generated from the phase locked loop are sent to the CPU via the PCM. By controlling the PCM register, clocks can be bursted to the CPU core from a minimum of 3% to the full 100% in steps of 3%. This is referred to as the CPU core duty cycle.

- While the CPU demand is low, for example in a calculator application, the clock can be bursted with a low duty cycle.
- While the CPU demand is high, for example in a handwriting recognition application, the clock can be running continuously at a 100% duty cycle.

The PCM uses a period of 32 clock cycles to burst the CPU core.

- For example, with a low duty cycle value of 12%, in any given period of time, the CPU core is active for 4 clock cycles (12% of 32 clock cycles), followed by 28 clock cycles of idle CPU core.

Please refer to the MC68328 User's Manual for full details on the operation of the PCM.

When using the PCM to control power management, the system clock from the Phase Locked Loop remains in high frequency. Since all peripherals on MC68328 are driven by the system clock, power saving on the CPU core can be achieved without sacrificing peripheral response time.

11.2 Power Modes

Figure 11-1 shows the state diagram for the power modes. There are six modes defined in PPSM.

System Internal Modes:

- Initialization mode
- System mode
- Wake-up mode

Application Modes:

- Normal mode
- Doze mode
- Sleep mode

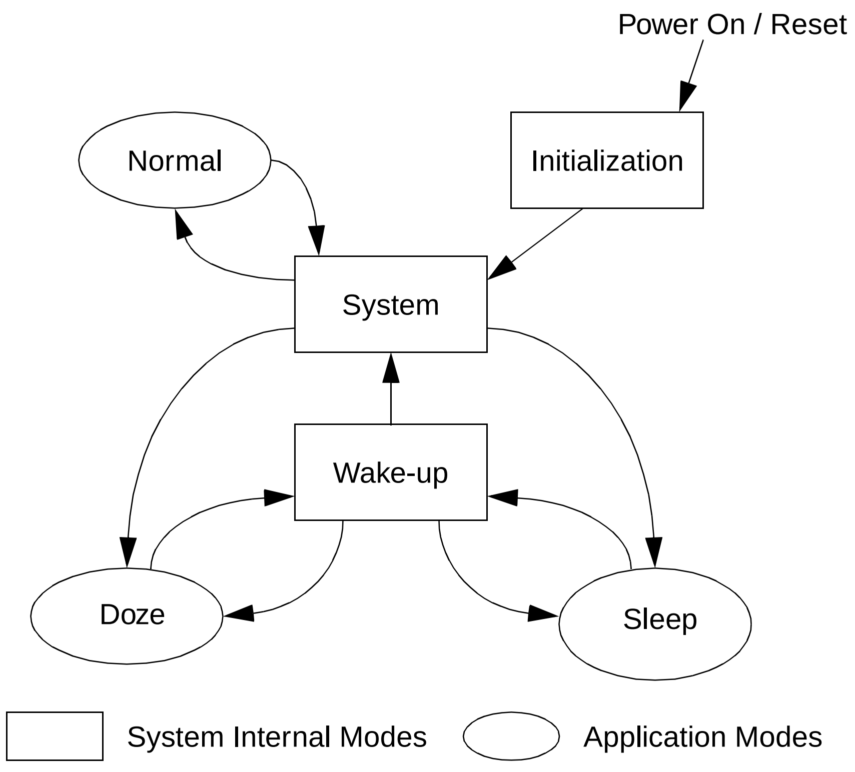


Figure 11-1 PPSM Power Modes

11.3 System Internal Modes

Initialization, System and Wake-up modes are only used internally by PPSM.

Applications need not be concerned with the three System Internal Modes. They are included in this chapter as a reference on the design of PPSM Power Management. All of these are intermediate modes among the Application Modes, where PPSM takes control to perform necessary system operations, such as task swapping, message passing and power management decisions.

11.3.1 Initialization Mode

This is the power on or system reset mode. Boot strapping and initialization of PPSM occur in this mode. The system never enters into this mode again once PPSM is initialized, unless system reset occurs.

11.3.2 System Mode

PPSM performs all of its task swapping, message passing, interrupt handling, power module controlling and modes switching in the System mode. This mode is frequently invoked, only for a very short duration, when the system is actively running, for example, to handle pen sampling, task swapping and message passing. To minimize the actual time spent in this mode, the duty cycle is set to 100% regardless of its set value prior to entering System mode. When the system leaves System mode, it will restore the duty cycle of the previous mode.

11.3.3 Wake-up Mode

This is invoked either from Doze or Sleep mode. When the system is in Doze or Sleep mode, only internal or external interrupts can wake up the system (please refer to *Section 11.4.2.3 - Waking up from Doze* and *Section 11.4.3.3 - Waking up from Sleep* for the Wake-up conditions).

In Wake-up mode, the system determines

- which of the interrupts occurred,
- where the interrupt messages, if any, should be sent to,
- which application, if any, needs rescheduling, and
- which mode the system should go into next.

For example, mid-night interrupt from the Real Time Clock, the system will:

1) Go into Wake-up mode from Sleep
2) Determine that the interrupt is intended for system only
3) Update the system date
4) Go directly back to Sleep mode

11.4 Application Modes

Normal, Doze and Sleep modes are the application modes. These are the modes that an application sees and can have control over.

11.4.1 Normal Mode

In this mode, applications can make use of the Power Control Module to control the CPU duty cycle value, please refer to *Section 11.5.1 - Setting Duty Cycle*. The Phase Locked Loop is on, all peripherals are active, LCD controller is enabled. Application is actively executing code.

PPSM is designed as an event driven system. It determines interrupt activities by monitoring the calls to the system tool IrptGetData().

- When there are interrupts, IrptGetData() returns to the application with interrupt messages. These messages are processed by the application accordingly.
- When there is no more interrupt pending for processing, a special message, IRPT_NONE is returned to the application.

11.4.2 Doze Mode

In this mode, the CPU is disabled to save power consumption. The LCD controller, Real Time Clock, Timer and Phase Locked Loop remain operational but all other peripherals are disabled. System is waiting for interrupts to wake up the CPU for more activities.

There are two ways for applications to enter Doze mode:

- direct system call
- automatic time-out

11.4.2.1 Direct System Call To Doze Mode

Application can go into Doze mode directly by calling SetDozeMode(), please refer to *Section 11.5.4 - Going Into Doze Mode*. In this operation, PPSM will put the system into Doze mode immediately, until a Wake-up condition is met.

11.4.2.2 Automatic Time-out To Doze Mode

Application can set a time-out period for the system to go into Doze mode from Normal mode. When PPSM detects that there are no more interrupt activities, either from the pen, timers, real time clock or external I/O, it will start this time-out period countdown. When this time-out expires, it will switch the system to Doze mode.

PPSM uses IRPT_NONE as an indication that the application is waiting for events and is ready to go into Doze mode. The doze time-out countdown, if set, begins.

However, if a Wake-up condition occurs before the doze time-out has expired, the doze time-out countdown is reset, and PPSM will return to the monitoring stage.

This automatic Doze mode time-out monitoring is repeatedly performed in Normal mode until the Doze mode period is set to zero, i.e. disabling Doze mode transition.

11.4.2.3 Waking up from Doze

Any one of the following internal or external interrupts can wake up PPSM from Doze mode in MC68328. The interrupts are:

` Pen Interrupt
` Real Time Clock Alarm, Periodic and Mid-night Interrupts
` Timer 1 and Timer 2 Interrupts
` External Interrupts from IRQ1, IRQ2, IRQ3, IRQ6, INT0-7

For MC68EZ328, whatever active interrupt before going to doze will be able to wake up the system.

The kind of interrupt to wake up the system can also be changed in PortDozeDisable() and restored in PortDozeEnable() in device driver.

The system can also be waked up by using SendMessage() or AdvSendMessage() to send message to the current task.

For Mid-night interrupt, system will wake up from Doze mode, update system date and time and then go back to Doze mode. For other interrupts existed above, system will wake up from Doze mode and go to Normal mode.

11.4.3 Sleep Mode

In this mode,

` CPU, Phase Locked Loop, LCD controller, and all peripherals are disabled.
` Only the Real Time Clock and Interrupt Handler Module are active.

This is the mode where power consumption is kept to a minimum as the most power consuming parts of the system, CPU and LCD, are off.

There are two ways for applications to enter Sleep mode:

` direct system call
` automatic time-out

11.4.3.1 Direct System Call To Sleep Mode

Application can go into Sleep mode directly by calling SetSleepMode(), please refer to *Section 11.5.5 - Going Into Sleep Mode*. In this operation, PPSM will put the system into Sleep mode immediately, until a Wake-up condition is met.

11.4.3.2 Automatic Timeout To Sleep Mode

Application can set a time-out period for the system to go from Doze mode to Sleep mode.

When PPSM puts the system into Doze mode, it will automatically start the Doze to Sleep time-out countdown, if set.

` When this time-out expires, PPSM will put the system into Sleep mode.

- If a Wake-up condition is met anytime during Doze mode before the time-out countdown for Sleep is reached, PPSM will reset the countdown and return the system to Wake-up mode.

This automatic Sleep mode time-out countdown is repeatedly performed in Doze mode until the Sleep mode period is set to zero, i.e. disabling Sleep mode transition.

11.4.3.3 Waking up from Sleep

Any one of the following internal or external interrupts can wake up PPSM from Sleep mode. The interrupts are:

- Pen Interrupt
- Real Time Clock Alarm, Periodic and Mid-night Interrupts
- External Interrupts from IRQ1, IRQ2, IRQ3, IRQ6, INT0-7, UART and PWM

For Mid-night interrupt, system will wake up from Sleep mode, update system date and time and then go back to Sleep mode. For other interrupts listed above, system will wake up from Sleep mode and go to Normal mode.

11.5 Power Management Tools

11.5.1 Setting Duty Cycle

U16 **SetDutyCycle**(U16 *percentage*)

This tool allows the application task to set the duty cycle level for itself in Normal mode. Applications within a system can have different duty cycle percentages. PPSM automatically changes the PCM accordingly when an application task becomes active.

11.5.2 Setting Doze Period

STATUS **SetDozePeriod**(U16 *millisecond*)

Sets the countdown period, in units of millisecond, to switch the system from Normal mode to Doze mode. A value of PPSM_NO_DOZE disables the system from going into Doze mode automatically which implies no automatically to sleep mode. A value of zero will bring back the system to default doze setting. The default doze setting is to go to doze mode whenever there is no task swap nor message in current task to be handled.

11.5.3 Setting Sleep Period

STATUS **SetSleepPeriod**(U16 *second*)

Sets the countdown period, in units of second, to switch the system from Doze mode to Sleep mode. A value of zero disables the system from going into Sleep mode.

11.5.4 Going Into Doze Mode

VOID **SetDozeMode**(VOID)

System goes directly to Doze mode. System will stay in Doze mode until a Wake-up condition is met.

11.5.5 Going Into Sleep Mode

VOID **SetSleepMode**(VOID)

System goes directly to Sleep mode. System will stay in Sleep mode until a Wake-up condition is met.

11.6 I/O Ports Control

For those I/O ports that are used by the hardware system, special handling will be required as PPSM does not have any knowledge of usage of these I/O ports. The system integrator will need to supply specific device routines that PPSM can call to disable and enable these I/O ports during Normal, Doze and Sleep mode transitions.

11.6.1 Disabling I/O Port Before Doze Mode

VOID **PortDozeDisable**(VOID)

Just before PPSM goes into Doze mode, it will call this routine to disable any user defined I/O ports that are not handled internally by PPSM. User must add in the code to disable the I/O ports in this routine.

11.6.2 Enabling I/O Port After Doze Mode

VOID **PortDozeEnable**(VOID)

When PPSM wakes up from Doze mode, it will call this routine to re-enable any user defined I/O ports that are not handled internally by PPSM. User must add in their own I/O initialization code in this routine.

11.6.3 Disabling I/O Port Before Sleep Mode

VOID **PortSleepDisable**(VOID)

Just before PPSM goes into Sleep mode, it will call this routine to disable any user defined I/O ports that are not handled internally by PPSM. User must add in the code to disable the I/O ports in this routine.

11.6.4 Enabling I/O Port After Sleep Mode

VOID **PortSleepEnable**(VOID)

When PPSM wakes up from Sleep mode, it will call this routine to re-enable any user defined I/O ports that are not handled internally by PPSM. User must add in their own I/O initialization code in this routine.

Chapter 12 UART Communication Support

PPSM supports serial communication through the UART in both normal mode and IrDA mode. A set of interface tools is provided for applications to send and receive data through the UART.

12.1 UART Communication Architecture

The UART interface tools provide an easy-to-use API for applications to send and receive data serially with or without hardware flow control. Refer to *Figure 12-1* and *Figure 12-2* for an overview of the UART communication architecture between a calling application and PPSM at system start up and during data transmission.

PPSM monitors the use of the UART among applications through IrptRequest() and IrptRelease() (refer to *Chapter 15 - Interrupt Handling*). The UART interface tools will have effect only after the calling application has been granted permission to access the UART. The data transmission is interrupt-driven.

Once permission is granted, the calling application can configure the UART, send or receive data through the UART, and be notified of the result of the send or receive operation. The same set of API tools is used for IrDA communication if the UART hardware is configured to run in IrDA mode.

12.1.1 UART hardware flow control

In PPSM v3.1, data communication between DragonBall and other communication devices using UART supports RTS, CTS hardware flow control. RTS is asserted automatically by calling UARTSend() and UARTReceive() when hardware flow control is enabled. In null modem configuration, when Dragonball is sender, receiver needs to acknowledge Dragonball ready to receive by asserting it□ RTS pin. When Dragonball is receiver, it acknowledges the sender side by asserting RTS pin. Thus, if both RTS pins of DragonBall and the other communication device are asserted, data transfer can be full-duplex.

Three APIs are available for RTS, CTS hardware flow control. They are UARTFlowCtrl(), UARTRcvCtrl() and UARTSendCtrl(). Hardware flow control can be enabled or disabled by calling UARTFlowCtrl(). By calling UARTRcvCtrl() and UARTSendCtrl(), PPSM can pause or continue data reception and data transmission respectively.

An API, UARTSendAbort(), is used for returning the current position of software send buffer and number of bytes have been transmitted by DragonBall. Also, this API can abort the transmission with appropriate input flag.

12.1.2 UART Interface Constraints

Only one task in the system can access the UART at any one time. Except for inquiring current settings, an error code will be returned to the application if it attempts to use the UART interface tools before permission is granted.

Applications swapping is inhibited during UART data transmission and reception. Once a data transmission or reception request has been initiated by an application, pen touches on application icons will be ignored. This constraint prevents data loss due to suspension of the application which initiated the request.

The enforcement of these constraints requires cooperation among applications in initiating a request only when it is needed, and cancelling a request as soon as data transmission or reception is completed.

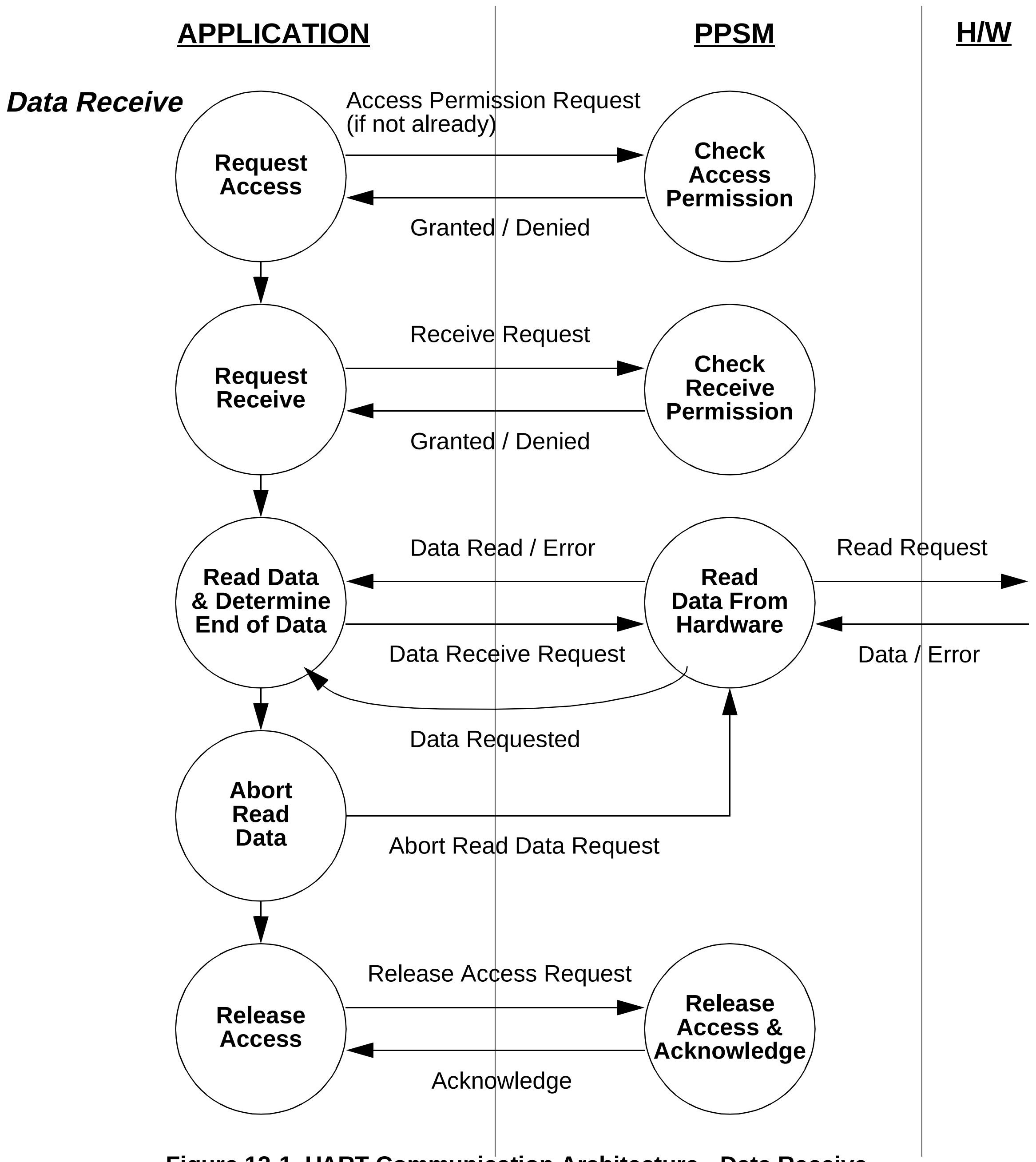


Figure 12-1 UART Communication Architecture - Data Receive

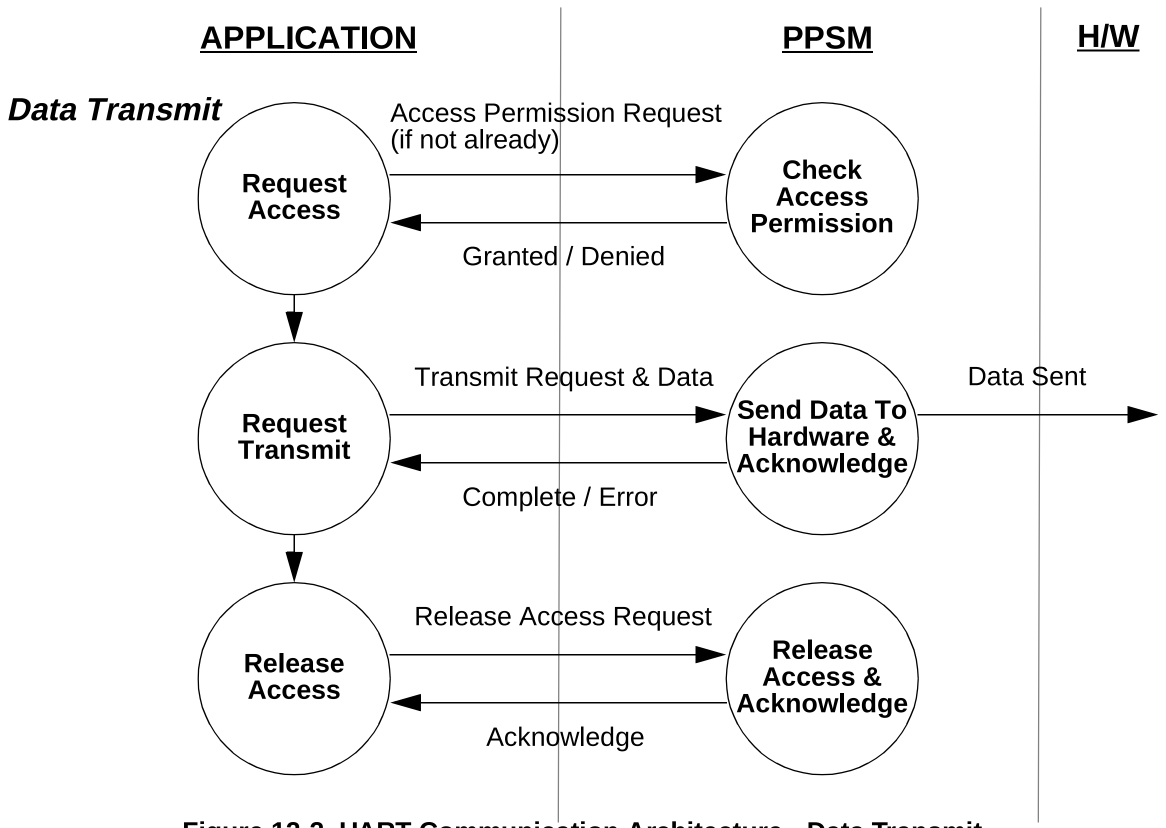


Figure 12-2 UART Communication Architecture - Data Transmit

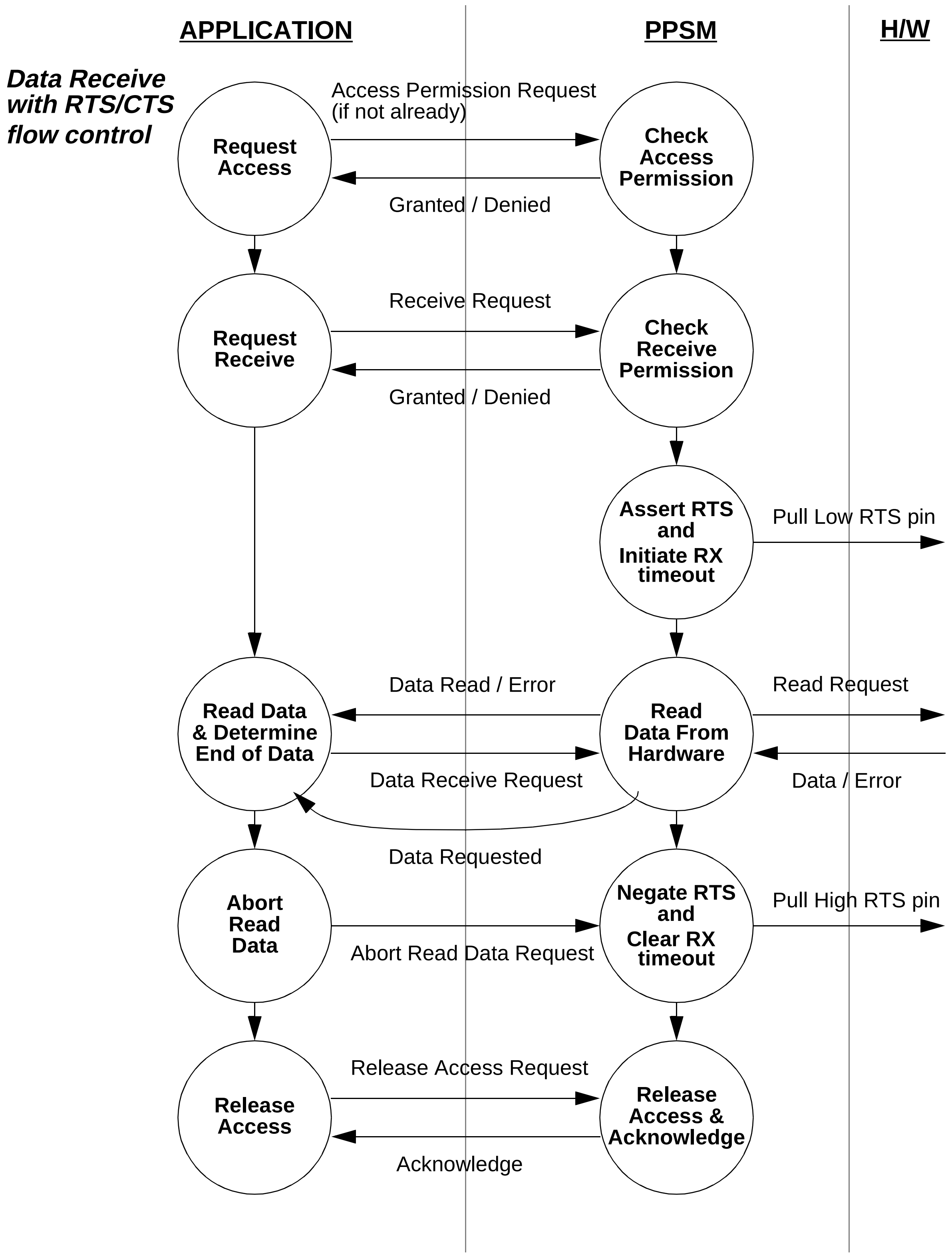


Figure 12-3 UART Communication Architecture - Data Receive with RTS/CTS flow control

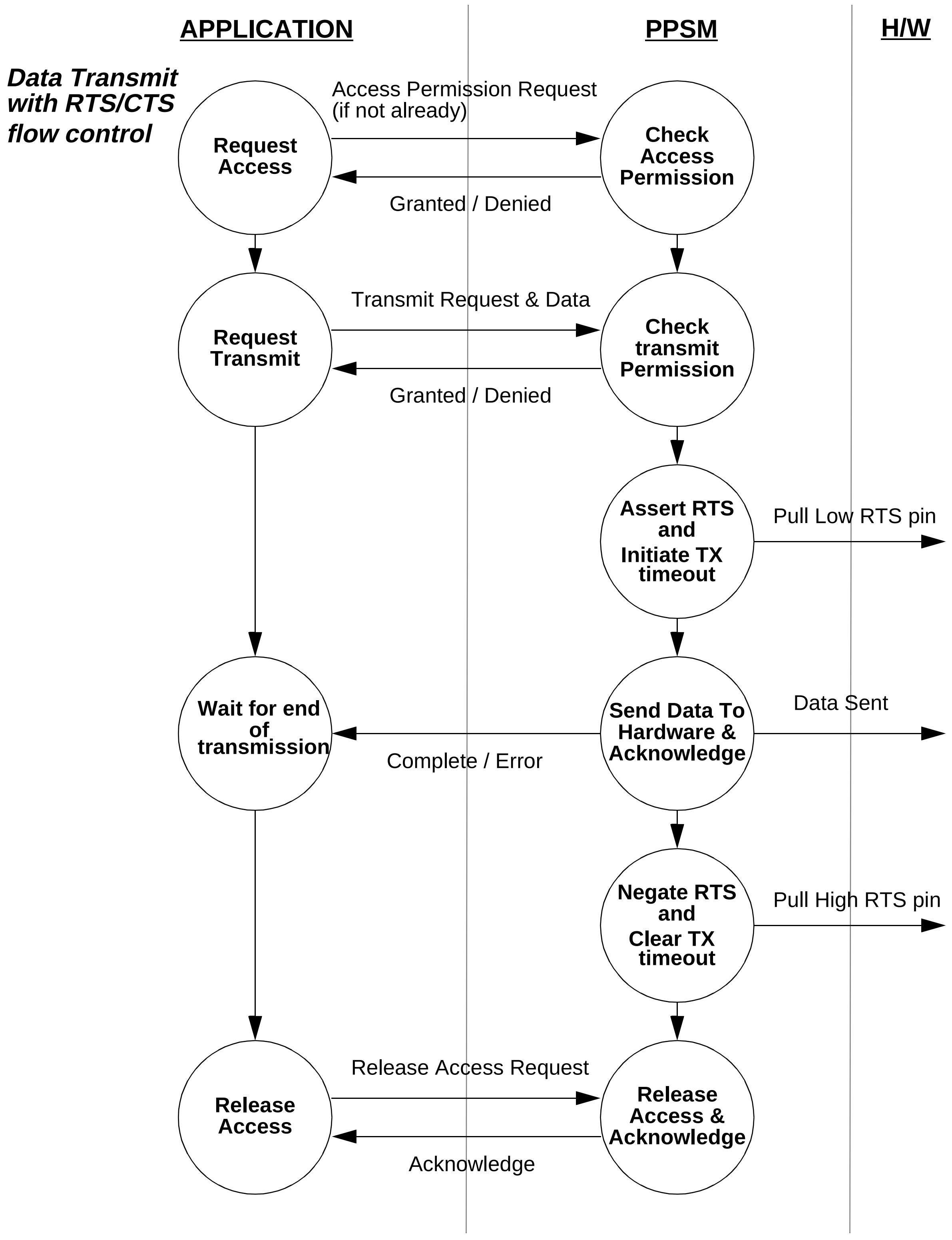

APPLICATION
PPSM
H/W
Data Transmit with RTS/CTS flow control
Request Access
Access Permission Request (if not already)
Check Access Permission
Granted / Denied
Request Transmit
Transmit Request & Data
Check transmit Permission
Granted / Denied
Assert RTS and Initiate TX timeout
Pull Low RTS pin
Wait for end of transmission
Send Data To Hardware & Acknowledge
Data Sent
Complete / Error
Negate RTS and Clear TX timeout
Pull High RTS pin
Release Access
Release Access Request
Release Access & Acknowledge
Acknowledge

Figure 12-4 UART Communication Architecture - Data Transmit with RTS/CTS flow control

12.1.3 UART Interface Interrupt Message

The UART interface communicates with an application via an interrupt message returned by IrptGetData(), called IRPT_UART. Please refer to *Chapter 15 - Interrupt Handling* for details about IrptGetData().

After an application is granted permission to use the UART, it can initiate a data transmission request. As the data transmission progressed, it will receive the IRPT_UART interrupt message with the corresponding message data under the following circumstances.

- An error condition has occurred. The interrupt message data, UART_ERROR, appended with an error code will be returned to the calling application. The error codes are:
  - UART_ERR_TMOUT for data transmission time out condition once the transmission has started.
  - UART_ERR_FRAME for frame error condition during data receive.
  - UART_ERR_PARITY for parity error condition during data receive.
  - UART_ERR_OVERRUN for overrun error condition during data receive.
  - UART_ERR_NODATA for prematurely requesting PPSM for data before data has been received.
- Data has been received from the UART. The interrupt message data, UART_DATA_RECEIVED, will be returned to the calling application.
- Data send request has been completed. The interrupt message data, UART_DATA_SENT, will be returned to the calling application.

Table 12-1 shows the new UART interrupt message and the related data returned with it during IrptGetData().

Table 12-1 UART Interrupt Message and related Message Data

Interrupt Message	Data Returned	Data Type
IRPT_UART	UART_ERROR followed by the actual error code: ` UART_ERR_TMOUT ` UART_ERR_FRAME ` UART_ERR_PARITY ` UART_ERR_OVERRUN ` UART_ERR_NODATA UART_DATA_RECEIVED UART_DATA_SENT	16-bit integer

12.2 UART Configurations

PPSM allows applications to configure the UART to operate in normal or IrDA mode, various baud rates, parity settings, stop bit settings, character length settings, and data transmission time out settings. When configured to operate in normal mode, the minimum and maximum baud rates supported are 300 bps (bits per second) and 115200 bps respectively. When configured to operate in IrDA mode, only the 115200 bps baud rate is guaranteed.

Upon system start-up, the default UART configuration is to run in normal mode at 9600 bps, with no parity, 8-bit characters, one stop bit, and no data transmission time out.

Please note that the application must have the permission to access the UART before it can configure the UART. Refer to *Chapter 15 - Interrupt Handling* regarding the usage of IrptRequest() to request permission.

12.2.1 Configuring the UART

STATUS **UARTConfigure**(U8 *mode,* U16 *baudRate*, U8 *parity*, U8 *stopBits*, U8 *charLen*)

Applications can use UARTConfigure() to reconfigure the UART to the required settings. Any on-going data transmission request will be aborted and the data transmission time out reset to the default. The actual baud rate will be the closest approximation to the specified baud rate.

Table 12-2 shows the list of configurations and settings supported, and the corresponding selection flag to be used with UARTConfigure(). (Refer to *Section 26.1 - UARTConfigure* for details)

Table 12-2 UART Configurations and Supported Settings

Configurations	Supported Settings
Operating Mode	` Normal NRZ mode ` IrDA mode
Baud Rate	` 300 bps ` 600 bps ` 1200 bps ` 2400 bps ` 4800 bps ` 9600 bps ` 14400 bps ` 19200 bps ` 28800 bps ` 38400 bps ` 57600 bps ` 115200 bps
Parity	` No parity ` Odd parity ` Even parity
Number of Stop Bits	` 1 stop bit ` 2 stop bit

Table 12-2 UART Configurations and Supported Settings

Configurations	Supported Settings
Character Length	` 7-bit character ` 8-bit character

12.2.2 Inquiring the UART Configurations

void **UARTInquire**(P_U8 *mode,* P_U32 *baudRate*, P_U8 *parity*, P_U8 *stopBits*, P_U8 *charLen*)

UARTInquire() provides the interface for applications to inquire the current configuration settings of the UART. UARTInquire() returns the selection flag for the corresponding configuration setting as shown in *Table 12-2*, except for baud rate. The actual baud rate in bps is returned instead.

Note: UARTInquire() is the only UART API tool that does NOT require UART access permission.

12.2.3 Setting Data Transmission Time Out

STATUS **UARTTimeout**(U16 *tmout*)

The data transmission time out is defined to be the time interval between two hardware UART interrupts. This time out is set to safe-guard the application from deadlocking itself when the data stream terminates unexpectedly.

If RTS/CTS is enabled, after called UARTSend() to initiate transmission, application will receive a time out error if CTS is not asserted within the time out period. If CTS is asserted, application will receive a time out error if the time interval between two hardware UART interrupts is larger than the time out period.

The range of time out values supported is zero to 60,000.

- Zero means disabling the time out function.
- 1 to 60,000 means allowing the time interval between two hardware UART interrupts to be 1 millisecond to 1 minute.

12.2.4 Setting Data Transmission Delay

STATUS **UARTSetDelay**(U8 *type,* U16 *delay*)

In order to communicate with application in PC, such as HyperTerminal and Telix, transmitting data in a burst of pulses periodically would greatly increase the accuracy of transmission. This function allows user to set a delay, in unit of 100us, between each transmission of all data in transmit FIFO (between two hardware interrupts). In *Example 12-1*, an application is going to transmit data through UART to HyperTerminal under Window95. A 400 microseconds delay is set between each UART hardware interrupt.

The range of delay values supported is 1 to 60,000.

- UART_TXDELAY_CLEAR means clear the delay during transmission
- 1 to 60,000 means allowing the delay interval between two hardware UART interrupts to be 100 microsecond to 6 seconds.

Example 12-1 Setting delay within transmission

```
.....
         IrptRequest(IRPT_UART_FLAG);
         /* Enable RTS/CTS flow control */
         UARTFlowCtrl(UART_RCTS_ENABLE);
         /* Configure UART */
         UARTConfigure( UART_NORMAL_MODE, UART_115200_BPS, NO_PARITY, ONE_STOP_BIT,
         EIGHT_BIT_CHAR);
         /* Set a 600 us delay between each hardware interrupt */
         _UARTSetDelay(UART_TXHALF_DELAY, 6);
         /* release irpt */
         IrptRelease(IRPT_UART_FLAG);
.....
```

12.3 Sending Data to the UART

STATUS **UARTSend**(U8 *sendFlag,* P_U8 *data,* U16 *dataLen*)

Refer to *Figure 12-2* and *Figure 12-4* for an overview of the data transmit architecture.

12.3.1 Initiating a Send Request

Applications can send data out to the UART by calling UARTSend() to initiate send requests. A send request will be accepted if both of the following are true:

- the application has permission to access the UART (refer to *Chapter 15 - Interrupt Handling*)
- there is no other on-going send request

Actual data sending does not happen within the scope of UARTSend(). If UARTSend() returns success for the request, PPSM will handle the UART interrupts and start sending data in the background. The application will be able to handle other interrupts (e.g. pen interrupts) in the foreground.

The calling application cannot modify the content of the data buffer during the entire course of the send request.

If RTS/CTS hardware flow control is enabled, PPSM only transmits data through UART when CTS pin is asserted by receiver.

For power saving reason, system is in Doze mode during transmission. However, transmission speed is reduced. For fast data transmission, it is recommended to disable the Doze mode before calling UARTSend() which is shown in *Example 12-2*.

Note: *Application swapping is disabled when there is an on-going data transmission.*

Example 12-2 Initiating a Send request

```
.....
        IrptRequest(IRPT_UART_FLAG);
        SetDozePeriod(PPSM_NO_DOZE);
        UARTSend(UART_SEND_REQUEST,gSendMsg,gSendDataLen);
.....
```

12.3.2 Terminating a Send Request

A send request will be terminated under the following circumstances:

` After PPSM finishes sending all data, it will post an IRPT_UART interrupt message with message data UART_DATA_SENT to the calling application. This marks the completion of the send request. (Refer to *Section 15.1.9 - IRPT_UART* for details about the UART interrupt message).

` If a timed out error condition occurs during the course of sending data, PPSM will post the IRPT_UART interrupt message with message data UART_ERROR and the corresponding error code. This marks a failed send request, and the calling application should determine the recovery actions. The current transmission is aborted after time out happened.

` An application aborts the on-going send request by calling UARTSend() or UARTSendAbort() with the abort flag.

The calling application should release the UART access permission by calling IrptRelease() as soon as it is not needed anymore.

It is recommended to force system to Doze mode after terminated a transmission or a transmission is completed which is illustrated in *Example 12-3*.

Note: Application swapping is re-enabled after a transmission is completed or aborted.

Example 12-3 Terminating a transmission

```
.....
        /* Abort send. Store the pointer of send buffer and no. of bytes have been..*/
        /* .. sent in gpSendbuf and gSendbyte respectively */
        UARTSendAbort(UART_SEND_ABORT, &gpSendbuf, &gSendbyte);
        IrptRelease(IRPT_UART_FLAG); /* release interrupt */
        SetDozePeriod(0); /* go to Doze mode */
.....
```

12.4 Receiving Data from the UART

STATUS **UARTReceive**(U8 *receiveFlag*)

STATUS **UARTReadData**(P_U8 *data*, U16 *bufSize*, P_U16 *sizeRead*)

Refer to *Figure 12-1* and *Figure 12-3* for an overview of the data receive architecture.

12.4.1 Initiating a Receive Request

Applications can receive data from the UART by calling UARTReceive() to initiate receive requests. A receive request will be accepted if both of the following is true:

- the application has permission to access the UART (refer to *Chapter 15 - Interrupt Handling*)
- there is no other on-going receive request

Actual data receiving does not happen within the scope of UARTReceive(). If UARTReceive() returns success for the request, PPSM will handle the UART interrupts and start waiting for data in the background. The application will be able to handle other interrupts (e.g. pen interrupts) in the foreground.

For power saving reason, system is in Doze mode during data reception. For fast data reception, it is recommended to disable the Doze mode before calling UARTReceive().

Note: Application swapping is disabled when there is an on-going receive request.

12.4.2 Reading Received Data

When PPSM has received data from the UART, it will post an IRPT_UART interrupt message with message data UART_DATA_RECEIVED to the calling application. The calling application should then call UARTReadData() as soon as possible to read the received data from PPSM.

As PPSM is receiving data from the UART, the following error conditions may arise:

- a frame error generated by the UART hardware
- a parity error generated by the UART hardware
- an overrun error when PPSM or the calling application is falling behind in reading the received data

In any of the above error conditions, PPSM will post the IRPT_UART interrupt message with message data UART_ERROR and the corresponding error code. These error related interrupt messages only serve as a notification to the calling application, and does NOT stop PPSM from continuing the receive request. The calling application should determine the appropriate recovery actions. (Refer to *Section 15.1.9 - IRPT_UART* for details about the UART interrupt message).

If RTS/CTS is enabled, RTS pin is negated when PPSM running UARTReadData() and asserted after data reading completed.

12.4.3 Terminating a Receive Request

A receive request will be terminated under the following circumstances:

- If a timed out error condition occurs during the course of receiving data, PPSM will post the IRPT_UART interrupt message with message data UART_ERROR and the corresponding error code.

This marks a failed receive request, and the calling application should determine the recovery actions. The current data reception is aborted after time out happened.

` An application aborts the on-going receive request by calling UARTReceive() with the abort flag.

The calling application should release the UART access permission as soon as it is not needed anymore.

It is recommended to force system to Doze mode after terminated a data reception or a reception is completed for power saving.

Note: Application swapping is re-enabled after a receive request is terminated.

12.4.4 Setting Data Reception Time Out

If RTS/CTS is enabled, after UARTReceive() is called, application may receive a time out error depends on whether CTS is asserted or not. If CTS is negated, application will not receive time out error because the other communication device is off. On the other hand, if CTS is asserted, application will receive time out error if no data arrived within the time out period. For the former case, application programmer is prefer to set a timeout by RefTimeAlarmId() after called UARTReceive() to avoid deadlocking as shown in *Example 12-4*.

Example 12-4 Setting initial time out for data reception

```
.....
IrptRequest(IRPT_UART_FLAG); /* request interrupt */
UARTTimeout(1000); /* set 1 sec time out between two receive interrupts */
UARTReceive(UART_RECEIVE_REQUEST); /* request to receive */
RefTimeAlarmId(&RxAlarmId, RefTimeRead()+2000); /* initiate 2 sec time out */
.....
      if(*inData==UART_DATA_RECEIVED)
      {     DeleteTimer(gRxAlarmId); /* data arrived, delete init time out */
            UARTReadData(gpBuf, gRcvBufSize, &gSizeRead); /* read data */
      .....
      }
```

12.5 UART hardware flow control

12.5.1 Enabling RTS/CTS hardware flow control

Applications can enable RTS/CTS hardware flow control by calling UARTFlowCtrl(UART_RCTS_ENABLE). If hardware flow control is not enabled when calling RTS/CTS flow control APIs, error code PPSM_ERR_RCTS_IDLE is returned to the application. RTS is asserted after enabled RTS/CTS flow control.

12.5.2 Disabling RTS/CTS hardware flow control

Applications can disable RTS/CTS hardware flow control by calling UARTFlowCtrl(UART_RCTS_DISABLE). System negates RTS pin immediately after disabled hardware flow control. Any further changes in RTS or CTS pin are ignored by the system. RTS is asserted after disabled RTS/CTS flow control.

12.6 Data reception with hardware flow control

12.6.1 Pause data reception

After RTS/CTS hardware flow control is enabled, PPSM automatically pauses data reception once internal UART buffer (not FIFO) is full. Data reception is resumed after data is read out by UARTReadData() in application. If the interval of CTS remain negated is longer than the time out period, time out error will occur.

Applications can pause data reception of UART when hardware flow control is enabled. Error code PPSM_ERR_RCTS_IDLE is returned if hardware flow control is not enabled. Applications can resume data reception by calling UARTRcvCtrl(UART_RCTS_CONT).

There will be no receive timeout error occur after pausing the data reception. Because the purpose of UARTTimeout( ) is mainly for avoiding the system stays in dead loop when transmitting or receiving data. User can set a timeout by calling RefTimeAlarmId( ). The receive timeout is restarted when data reception is resumed by calling UARTRcvCtrl(UART_RCTS_CONT).

12.6.2 Continue data reception

Applications can continue data reception which has been paused by UARTRcvCtrl(UART_RCTS_PAUSE) when hardware flow control is enabled. Error code PPSM_ERR_RCTS_IDLE is returned if hardware flow control is not enabled.

12.7 Data transmission with hardware flow control

12.7.1 Pause data transmission

Applications can pause data transmission of UART when hardware flow control is enabled. Error code PPSM_ERR_RCTS_IDLE is returned if hardware flow control is not enabled. Applications can resume data transmission by calling UARTSendCtrl(UART_RCTS_CONT).

There will be no transmit timeout error occur after pausing the data transmission. Because the purpose of UARTTimeout( ) is mainly for avoiding the system stays in dead loop when transmitting or receiving data. User can set a timeout by calling RefTimeAlarmId( ). The transmit timeout is restarted when data transmission is resumed by calling UARTSendCtrl(UART_RCTS_CONT).

12.7.2 Continue data transmission

Applications can continue data transmission which has been paused by UARTSendCtrl(UART_RCTS_PAUSE) when hardware flow control is enabled. Error code PPSM_ERR_RCTS_IDLE is returned if hardware flow control is not enabled.

Chapter 13 Task Management

Each application running on PPSM is considered as a task. There are two types of PPSM tasks:

` main task - application task that is stand alone
` sub-task - task that is spawned off by another task either main or sub task.

The task management tools enable applications to:

` Create a main task or a sub-task
` Start execution of a task
` Terminate execution of a task

This chapter describes how applications can make use of PPSM tools to generate PPSM tasks.

Message passing or task swapping are possible among any sub tasks, its parent task and any other main tasks. Changing of panning screen in main task will affect the panning screen parameter of its sub tasks. Changing of panning screen in sub task will affect the panning screen parameter of its main task and other sub tasks belonging to the same main task.

13.1 Main Task

Most applications fall into the main task category. Main tasks run independently of each other. There cannot be more than 1 main task running at any given time. They are created by the system tool TaskCreate() or AdvTaskCreate(). Once a main task is created, there are three ways it can be started:

` By using the system tool TaskStart()
` By pressing the application icon
` By messages sent from another task

13.1.1 System Task

System task is a special main task which is created in PPSMInit(). It□ never terminated since creation. The stack used in this task comes from the stack defined in SPC file so it cannot be freed. The panning screen created in this task can be deleted and replaced by another global panning screen by the following method:

Example 13-1 Sharing system task□ panning screen using global variable

```
/* Global variable for sharing panning screen */
U32 gPanScreen;

STATUS TaskApp()
```

```
{
      PAN_SCREEN tempScreen;

      tempScreen.panAddress = tempScreen.displayScreenAddr = gPanScreen;
      tempScren.horzSize = 160;
      tempScreen.vertSize = 240;
      tempScreen.displayXOrigin = tempScreen.displayYOrigin = 0;
      tempScreen.regPOSR = 0;
      tempScreen.regPSW = tempScreen.horzSize/PIXELS; /* where PIXELS=8 if the LCD
      is set to 2 bits/pixel and PIXELS=16 if the LCD is set to 1 bit/pixel. */

      /* Set the panning screen of current task to the sharing panning screen */
      ChangePanning(tempScreen, 0);

      while(1)
      {
            ....
      }
}

main()
{
      U32 taskAppId;
      PAN_SCREEN tempScreen;

      /* PPSM Initialization */
      PPSMInit(FALSE);

      /* Get screen memory */
      gPanScreen = (U32)GetScreenMem(160, 240);

      /* Assign panning screen parameters */
      tempScreen.panAddress = tempScreen.displayScreenAddr = gPanScreen;
      tempScren.horzSize = 160;
      tempScreen.vertSize = 240;
      tempScreen.displayXOrigin = tempScreen.displayYOrigin = 0;
      tempScreen.regPOSR = 0;
      tempScreen.regPSW = tempScreen.horzSize/PIXELS; /* where PIXELS=8 if the LCD

      /* Delete the default system panning screen and assign the new panning screen
      to system task */
      ChangePanning(tempScreen, 0);

      /* Create a main task without application icon nor panning screen */
      AdvTaskCreate(taskAppId, TaskApp, 0, 0, 0, 0, 2048, PPSM_SCREEN_NOSCREEN, 0,
      0, NULL);

      TaskStart(taskAppId);
}
```

13.2 Sub-task

Sub-task, on the other hand, can be active at the same time as the parent task that generated the sub-task and other sub tasks with same parent task. A main task can create multiple sub-tasks. These sub-tasks are queued in the reverse order they are created initially. However, the order may be changed when any sub task is swapping in or out by using SendMessage() or AdvSendMessage().

Sub-task uses the display resource, hardware cursor and input pad of its parent and can only be created with the system tool SubTaskCreate().

Sub-tasks are tied to the parent task. If the parent task is swapped out or terminated, the sub-task will be swapped out or terminated too. Sub-task inherits the input pad properties from the parent task at creation. There can only be one input pad among the main task and its sub-tasks.

13.2.1 Sub-task Management

When the active area of sub task is touched, that sub task will be swapped in as the current task.

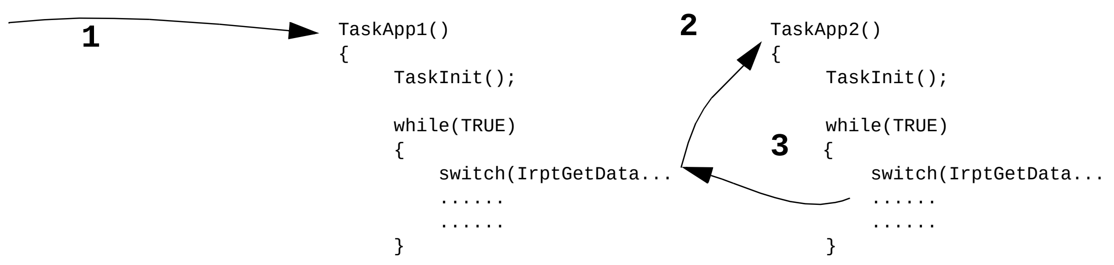
The IrptGetData() tool is another task swapping point. If no interrupt message is pending among a main task and its sub-task when IrptGetData() is called, the current main or sub-task remains active. If message is pending in any of the main task or its sub tasks, task swapping will happen in IrptGetData().

If there are multiple sub-tasks, they will be parsed using round-robin method. The most recently swapped out sub-task will be put to the end of the queue. When the system is ready to restart a new sub-task, it always searches from the beginning of the queue.

13.3 Task Switching

When the task is started for the first time, it will execute from the beginning of the task application. When the task needs to be swapped out, PPSM will save the current Program Counter value. Then, when this task is swapped back in, it will resume execution from where it was left off.

Example 13-2 Task switching



```
TaskApp1()                              TaskApp2()
{                                       {
    TaskInit();                             TaskInit();

    while(TRUE)                             while(TRUE)
    {                                       {
        switch(IrptGetData...                   switch(IrptGetData...
        ......                                  ......
        ......                                  ......
    }                                       }
```

At arrow 1, TaskApp1() is started for the first time. Then, at arrow 2, TaskApp2() is also started for the first time, so TaskApp1() will be swapped out. At arrow 3, TaskApp1() is swapped back in and is resumed from where it was left off.

If the task is swapped by pen interrupt or AdvSendMessage() with input parameter being SWAP_TASK or SWAP_TASK_BACK_LATER.

If the task is swapped by SendMessage() or AdvSendMessage() with input parameter being SWAP_TASK_LATER, the current task will not be swapped out immediately. It will be swapped out in IrptGetData() when all messages in current task are handled. If the target task is a subtask in other family, it will swap to the parent of the other family before swapping to the target subtask.

All tasks to be swapped in will be in a FIFO queue. The current task to be swapped out will be put at the end of the queue. However, if the current task needs to be swapped in later by AdvSendMessage() with SWAP_TASK_BACK_LATER, the current task will be put in the head of the queue while it□ swapped out.

If the current task is parent task and there is no more message to be handled, the system will check whether there is need to swap to next parent task in the queue. If there is no more main task to be swapped in, the system will check from the head of subtask to see whether any subtask needs to be swapped in. The subtask to be swapped in may due to message in the queue or the task swapping flag in the subtask is on.

If the current task is subtask and there is no more message to be handled, the system will check whether there is need to swap to the next subtask in the head of subtask queue. If not, system will check for the parent and then rest of the subtask to see whether they need to be swapped in or they have message to handle. If no more message or task to swap in, system will check the head of the main task queue to see whether it needs to be swapped in.

Task switching can be disabled by calling AppSwap(FALSE). AppSwap() is a function to stop task swapping while the system is transferring UART data or any other critical operations. If the AppSwap(FALSE) is called several times, the same number of times of AppSwap(TRUE) must be called to let the task switching active again.

13.4 Message Broadcasting

If the application programmer stores the task id, including main and sub tasks, into a global list, message can be broadcasted to this list of task by using AdvSendmessage() with or without task switching.

13.5 Task Control

If the application programmer stores the task id, including main and sub tasks, into a global list, the task swapping sequence can be controlled by using AdvSendMessage() with or without message passing. However, those task on the task swapping queue will not be affected. So if a task is already on the task swapping queue, nothing can be used to change it. For those task not being on swapping queue, AdvSendMessage() can be used to put it into the task swapping queue. The earlier the task is put into the task swapping queue, the higher priority the task will be swapped in.

In task swapping, PPSM will check whether there is other main task to swap to before checking the sub tasks of current main task.

The system will always check to see whether their are any messages need to be handled within the family and swap to that member task to finish the job. Sometimes the message in current task are cleared and SWAP_TASK is called immediately to other member of the family. This task may be swapped bad later as the memory for the last message is not free yet. Message is still in the task until next IrptGetData() is called. And next task swapping for SWAP_TASK_LATER in that task will be next IrptGetData() after the one freeing last message□ memory.

13.6 Task Swapping Example

```
TaskAppA()
{
      SubTaskCreate(&subTaskA1, ...);
      SubTaskCreate(&subTaskA2,....);
      AdvSendMessage(taskB, 0, SWAP_TASK_BACK_LATER);
      .....
      while(1)
      {
            IrptGetData(...);
            ....
      }
}

TaskAppB()
{
      SubTaskCreate(&subTaskB1,...);
      SubTaskCreate(&subTaskB2, ...);
      .....
      While(1)
      {
            IrptGetData(....);
            .....
      }
}

SubTaskAppA1()
{
      .....
      While(1)
      {
            IrptGetData(....);
            .....
      }
}

SubTaskAppA2()
{
      .....
      AdvSendMessage(taskB, 0, SWAP_TASK);
      .....
      While(1)
      {
            IrptGetData(....);
            .....
      }
}

SubTaskAppB1()
{
      .....
      While(1)
      {
            IrptGetData(....);
            .....
      }
}

SubTaskAppB2()
{
      .....
      AdvSendMessage(taskB, 0, SWAP_TASK_LATER);
      AdvSendMessage(taskA1, 0, SWAP_TASK_LATER);
      AdvSendMessage(taskB1, 0, SWAP_TASK_LATER);
      ......
      While(1)
      {
            IrptGetData(....);
            .....
      }
}
```

Task swapping sequence is A->B->A->A1->A2->B->B1->B2->B1->B->A->A1

When task A is created, it will create subtask A1 and subtask A2. These 2 subtasks wouldn□be executed until IrptGetData() is called. However, AdvSendMessage() with SWAP_TASK_BACK_LATER is called before IrptGetData() so the next task is B. In task B, subtask B1 and subtask B2 are created. In IrptGetData() of task B2, it will swap back to A as the previous command is SWAP_TASK_BACK_LATER. Then subtask A1 and subtask A2 will be executed in sequence. In subtask A2, SWAP_TASK to B is executed so next task is B. In IrptGetData() of task B, it will swap to B1 as B1 is not executed yet. Then in IrptGetData() of subtask B1, it will swap to subtask B2. In subtask B2, SWAP_TASK_LATER is called for task B, subtask A1 and subtask B1. As the system will check for next subtask first, subtask B1 will be swapped in IrptGetData() of subtask B2. In IrptGetData() of subtask B1, it will swap to B as it will check for the parent after checking the next subtask. Then it will swap to task A and then A1 in IrptGetData() of these tasks as AdvSendMessage() is called for swapping task to A1 in subtask B2. Whenever SWAP_TASK_LATER is called for subtask in other family, the parent of the other family will be swapped in first.

13.7 Creating a Task

STATUS **TaskCreate**(P_U32 *taskId*, P_VOID *procAddr*, S16 *xSrc*, S16 *ySrc*, S16 *xDest*, S16 *yDest*, P_U8 *bitmap*)

PPSM needs to know the existence of each application task before the task can access PPSM resources. The main body of a PPSM system must call this routine once for each application. PPSM will create the necessary data structure and memory space required to run the application. An application icon is created for each application with the coordinates as supplied in the argument list. The application is put to the foreground whenever this icon is selected. This tool does not start the execution of the application. It registers the task with PPSM only. If the user does not want to have an application icon, the user should set either width or height to be zero(*xSrc* = *xDest* or *ySrc* = *yDest*). Hence, there is no application icon to be created.

By default, a screen is created with the task. PPSM uses the system default physical size as the dimension for this screen. The default physical size is specified in the Linker Specification File, as described in *Chapter 34 - Linker Specification File*.

A 2K byte of memory is allocated for each task as the task□ stack.

13.8 Creating a Task with Specific Task Parameters

STATUS **AdvTaskCreate**(P_U32 *taskId,* P_VOID *procAddr,* S16 *xSrc,* S16 *ySrc,* S16 *xDest,* S16 *yDest,* S32 *stackSize,* U16 *newScreen,* U16 *screenWidth,* U16 *screenHeight,* P_U8 *bitmap*)

Creation of a new PPSM task. This tool creates a PPSM application task in the same manner as the existing tool TaskCreate(), with the difference that it also allows the caller to specify the launch icon position and size, the stack memory required by the application and the screen memory size, if any is required. Two settings for the panning screen variable, newScreen, are available:

- PPSM_SCREEN_NOSCREEN will have no screen.
- PPSM_SCREEN_NEW will take the arguments *screenWidth* and *screenHeight* and creates a new screen for the application task. However, if either one of the arguments, *screenWidth* and *screenHeight*, is zero, the default panning screen size taken from the linker specification file will be used.

A default of 512 byte of memory is allocated for the task□ stack if the input argument is negative.

Example 13-3 Create a task

```
main()
{
  U32   SlideTask;                /* Task id for slide                   */
  U32   UartDemoTask, TimerTask;  /* Task id for uart and reference timer */
.
.
.
  /*  Create the UART application task with a stack size = 2K,
   *  and a panning screen with default width & height is required.
   */
  if (AdvTaskCreate(&UartDemoTask, (P_VOID) UartDemo, src_x[UART_ICON],
    src_y[UART_ICON], dest_x[UART_ICON], dest_y[UART_ICON], 2048,
    PPSM_SCREEN_NEW, 0, 0, 0))
    return(PPSM_ERROR);
```

13.9 Creating a Sub Task

STATUS **SubTaskCreate**(P_U32 *taskId*, P_VOID *procAddr*, U16 *stackSize*, U16 *numArg*, ...)

Creating a sub-task. Any task can use this tool to create sub-tasks. If the calling task is itself a sub-task, the new sub-task will belong to the calling sub-task□ parent(ie. the calling and the created sub-task will become siblings). If the calling task has already created more than one sub-task, the new sub-task will be added to the head of the sub-task list. There is currently no limit on the number of sub-task a parent task can create.

This routine accepts variable length input argument. These arguments are passed into the sub-task by PPSM, meaning that the actual sub-task routine can accept input arguments.

Subtask will be started when the current task has no more messages to be handled and there is no need to swap to the other task. Subtask will be started in the IrptGetData() routine in current task.

13.10 Starting a Task

STATUS **TaskStart**(U32 *taskId*)

This tool launches a task that has been created by the tool TaskCreate() or AdvTaskCreate(). This routine never terminates. It takes the task identifier as input argument and begins execution of the task. Since this tool may prevent the application caller from executing again, it should always be called at the end of main() to start the first task.

Example 13-4 Start a task

```
U32   SlideTask;                /* Task id for slide                    */
.
.
.
/* Slide is the default task to be run when the system starts up */
TaskStart(SlideTask);
```

13.11 Termination of a Task

STATUS **TaskTerminate**(U32 *taskId*)

Termination of a task. The task identifier can be of a main or sub-task. All system memory associated with the task and its sub-tasks that are allocated by PPSM are freed, such as stack memory and screen, if any. Any memory that was explicitly allocated by the task through Lmalloc(), Lcalloc() or Lrealloc(), is not going to be freed by the system because that area may be shared by several tasks. That area can be freed by calling Lfree() in application program.

A task cannot terminate itself. If it is a sub-task, it cannot terminate its parent task either.

13.12 Task Reinitialization

STATUS **TaskReInit**(U32 *taskId, U16 flag*)

This tool will set the reinit flag in the specified task. If the flag is TRUE, whenever the task is swapped in, it will start at the beginning of the task function. However, application programmer needs to handle the cleaning up of memory in task swapping using TaskHook().

In task swapping, the PC and stack, etc. will be restored to the value when the task is not executed.

This function must be called immediately after TaskCreate() or AdvTaskCreate() when the task is created. This function can be called to disable the task reinitialization at anytime but cannot be called to enable the reinitialization again.

13.13 Task Hook

STATUS **TaskHook**(U32 *taskId,* P_VOID *entryCallback*, P_VOID *exitCallback*)

This tool will hook the entryCallback() and exitCallback() functions to the specified task. When the task is swapped in, the entryCallback() will be called after updating the registers. When the task is swapped out, the exitCallback() will be called before storing the registers value. The *entryCallback* and *exitCallback* must be an one input parameter function. The input parameter for *entryCallback* will be the task id. for the task just swapped out and the input parameter for *exitCallback will be the task id.* for the task swapping in.

e.g. STATUS Entry(U32 previousTaskId); and STATUS Exit(U32 nextTaskId)

This function can be called outside or within the task. However, if it□ called within the task, the entry routine is not executed and so it needs to be called after this function.

Example 13-5 TaskHook() inside task

```
VOID EntryR(U32 oldId)
{
......
}

VOID ExitR(U32 nextId)
{
......
}

Task1()
{
      Task1Init();
      TaskHook(task1Id, EntryR, ExitR);
      EntryR(0);

      While(1)
      ....
}
```

If TaskHook() is called outside its task, the entry routine will be executed automatically once the task is executed.

13.14 Stop task swapping

void **AppSwap**(U16 *flag*)

If *flag* is FALSE, no task swapping will be allowed by any means. If SendMessage() or AdvSendMessage() is called after calling AppSwap(FALSE), the message will be sent but no task swap later nor task swap immediately will be executed. This function will increment a flag for task swapping, if it□ called with FALSE several times, the same number of times AppSwap(TRUE) must be called before task swapping is allowed.

Chapter 14 Inter-Task Messaging

PPSM supports asynchronous message passing between tasks using the provided tools. This tool cannot be used to pass message between sub tasks with different parent tasks.

The sender task sends out the message stored in the pre-defined message structure using the tool SendMessage() or AdvSendMessage(). It must know the task identifier of the task that it wants to send the message to.

The receiving task receives the sent message in it□ software interrupt buffer, in the same way as other interrupt messages sent from PPSM system. Accessing this message by the application is done by using the IrptGetData() tool.

The messaging tool enables applications to:

` notify other applications of user defined events
` pass data between tasks within the system
` swap to the specified task immediately or later

The format of the data passed by these tools are not defined. The caller and receiver must have their own mutual agreement on the form of data being sent. PPSM only performs the actual message passing and informing the receiving application task of the arrival of the message.

The AdvSendMessage() can be used to control task swapping sequence with or without message passing.

This can be used to send message to the task itself. Whenever these functions are called to send message or control task swapping in interrupt routines when the system is in doze mode, it will wake up the system.

14.1 Message Passing

Message can be sent between any tasks even the current task itself. SendMessage() will send message to the target task and set the flag to swap to the target task once all messages for current task are handled. AdvSendMessage() has more flexibility. Task swapping can happen without any message sent to the target task using AdvSendMessage(). AdvSendMessage() can be used to send message to target task without task swapping which is used as a purely message passing tool.

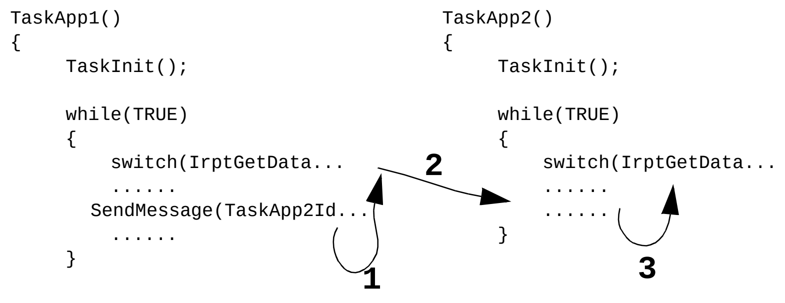
Example 14-1 Multiple swap task later

```
TaskA()
{
      ......
      SendMessage(taskBId, msg);
      SendMessage(taskCId, msg);
      .....
}
```

In the above example, task B will be put on the head of task swapping queue and then task C will be put after task B. So when all messages in task A are handled, system will swap to task B. When all messages in task B are handled, system will swap to task C.

> *Note: If UARTSend() or UARTReceive() is called, no task swapping can be happened in any case until transmission or reception is aborted. It is applied to all task swapping cases discussing in this chapter.*

14.1.1 With Delayed Task Swapping



```
TaskApp1()                              TaskApp2()
{                                       {
     TaskInit();                             TaskInit();

     while(TRUE)                             while(TRUE)
     {                                       {
         switch(IrptGetData...                   switch(IrptGetData...
         ......                                  ......
       SendMessage(TaskApp2Id...                 ......
         ......                              }
     }
```

In the diagram above, TaskApp1() will call SendMessage() to send a message to TaskApp2(). However, the active TaskApp1() will not be swapped out. It will still be active until it executes IrptGetData() as in arrow 1. Then, TaskApp2() will be swapped back in as in arrow 2. Message from TaskApp1() will be received in IrptGetData() of TaskApp2() as in arrow 3.

The task swapping happens when all messages in TaskApp1() are handled.

This will happen if SendMessage() or AdvSendMessage() with SWAP_TASK_LATER are used. The target task will be push into the tail of main task swapping queue if the target task is main task or the tail of sub task swapping queue otherwise.

14.1.2 With Immediate Task Swapping

In AdvSendMessage(), if the flag is SWAP_TASK, message passing and task swapping will happen inside AdvSendMessage() immediately.

14.1.3 With Immediate Task Swapping and Delayed Swap Back

In AdvSendMessage(), if the flag is SWAP_TASK_BACK_LATER, message passing and task swapping will happen immediately inside AdvSendMessage() and the current task will be put into the head of the task swapping queue. So when all messages are handled in target task, the current task will be swapped back.

14.1.4 Message Passing without Task Swapping

In AdvSendMessage(), if the flag is NO_SWAP_TASK, it will act as a message passing only and no task swapping activity will happen.

14.2 Message Structure

A pre-defined structure is used to store and forward messages from the sender.

```
typedef struct _MESSAGE
      {
            U16    messageType;        /*  message type  */
            U16    message;            /*  message  */
            U32    misc;               /*  short data (32bit)  */
            P_VOID data;               /*  associated data, if any */
            U16    size;               /*  size of data in bytes  */
            U16    reserved;           /*  for future  (broadcast, etc.) */
      } PPSM_MESSAGE, *P_MESSAGE;
```

Table 12-1 Message Structure

Name	Description
messageType	The type of message being sent. Currently only one type of message is defined for application usage. This is MESSAGE_IRPT. Application MUST set this field to MESSAGE_IRPT, otherwise, the message will not be sent.
message	The interrupt message. This is passed directly to the receiving application as the return value when the IrptGetData() tool is called. Default value should be set to IRPT_USER, as a user-defined interrupt type. Application developers can set their own 16-bit value. See *Section 14.7 - Receiving Message* for details.
misc	Unformatted 32-bit value. This is passed directly to the *sData* field of the IrptGetData() tool.
data	Pointer to any data that might be passing from the sender to the receiver. The data type and format is user defined. PPSM does not put any protocol into this data format.
size	Size of the data in the data pointer list. This is in number of bytes.

14.3 Sending Message

STATUS **SendMessage**(U32 *taskId*, P_MESSAGE *msg*)

This tool sends a message to a known task. If the receiver task□ task identifier is not known, this tool cannot be used.

All data that the sender wants to send must be stored in the form of MESSAGE structure. No protocol or data format is put on the message data. The sender and receiver must have a mutual understanding of the representation of the data being transferred.

14.4 Advanced Sending Message

STATUS **AdvSendMessage**(U32 *taskId*, P_MESSAGE *msg,* U8 *flag*)

This is similar to SendMessage() except it enhances the task swapping control. The *flag* can control whether the target task is swapped in immediately or later. It also controls whether the current task will be swapped in again after all messages in target task are handled.

If *msg* is NULL, this is a task swapping control function without message passing.

If *flag* is NO_TASK_SWAP, this is a message passing tool without task swapping control.

14.5 Deleting Message for Current Task

STATUS **MessageDelete**(U16 *type*)

This is for deleting all messages in current task with the same type as the input parameter such as IRPT_PEN, IRPT_UART, etc.

14.6 Deleting Message for any Task

STATUS **AdvMessageDelete**(U32 *taskId*, U16 *type*, U32 *shortData*)

This is for deleting messages in specific task matching the *type* and *shortData*. The short data here refers to the area id. for active area, timeout id. for reference timer, etc. If *taskId* is 0xFFFFFFFF and the current task is a main task, all messages in main task queue with matching *type* and *shortData* will be deleted. If the taskId is 0xFFFFFFFF and the current task is a subtask, all messages with matching *type* and *shortData* in current sub task list will be deleted. If the type is 0xFFFF, all messages with matching *taskId* and *shortData* will be deleted.

If *type* is 0xFFFF and *shortData* is 0xFFFFFFFF, all messages in the specific task will be deleted.

14.7 Receiving Message

U16 **IrptGetData**(P_U32 *sData*, P_U32 **data*, P_U32 *size*)

This tool is used to receive the messages sent by another task, as well as to receive the standard software interrupt message (see *Section 29.1 - IrptGetData*).

The arguments returned by this tool can be mapped directly to the data stored in the MESSAGE structure sent by the SendMessage() tool. They are as follows:

IrptGetData	Data Type	Data field in MESSAGE structure	Data Type
return value	U16	message	U16
sData	P_U32	misc	U32
data	P_U32*	data	P_VOID
size	P_U32	size	U16

Example 14-2 Receive messages

```
STATUS UartDemo()
{
  P_U16         inData;
  U8         selected;
  U32        size, id;
    .
    .
    .
  while (1)
    {
      switch (IrptGetData( (P_U32)&id, (P_U32*)&inData, (P_U32)&size))
       {
       case IRPT_UART:
         switch (*inData)
           {
           case UART_DATA_RECEIVED:

             /* Data has been received, read data from system */
```

Chapter 15 Interrupt Handling

PPSM maintains a set of interrupt handlers internally to handle external and internal hardware events. PPSM application programmers do not need to be aware of the characteristics of hardware such as pen device, timer, UART and real time clock. The kernel intercepts all interrupts and data generated from the event and send them to the application in a pre-defined format.

PPSM maintains a unique software interrupt buffer for each application. When a hardware interrupt occurs, the event and data generated from the interrupt are stored into this software interrupt buffer. This buffer is the interface between PPSM and the application, making sure that data and interrupt will not be missed if the application is slow in response to an interrupt. Hence de-coupling the real-time interrupt of the peripheral devices from the application.

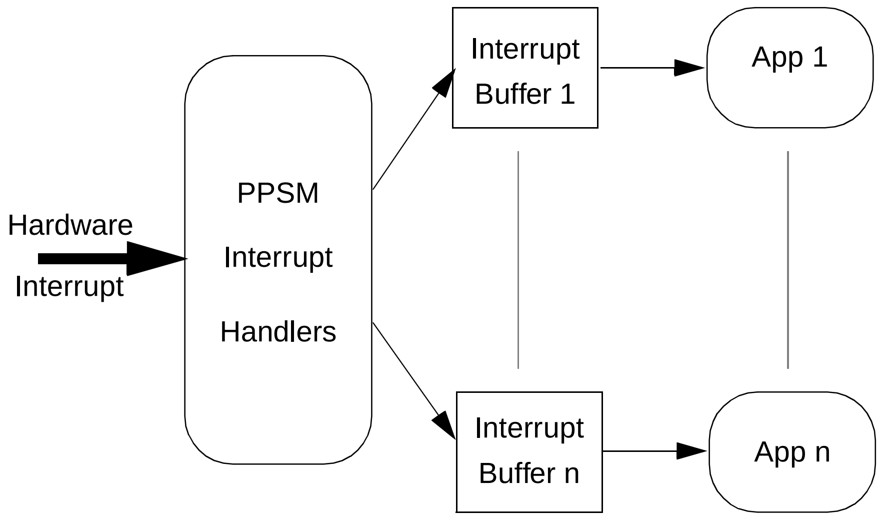


Figure 15-1 PPSM Interrupt Message Handling

PPSM has two distinct types of interrupts:

- System interrupt
- User defined interrupt

15.1 System Interrupts

These are interrupts that are automatically handled by PPSM system. Application developers can make use of the services provided by PPSM to manage the hardware resources such as the touch panel, timer, UART, RTC and screen.

Table 15-1 shows the list of interrupt identifiers that are generated by the system to the application.

Table 15-1 System Interrupt Identifiers

Interrupt Identifier	Interrupt Source	Data from Handler
IRPT_AUDIO	PPSM Audio tools, indicating audio stopped	N/A
IRPT_HWR	Handwriting recognition	Data from handwriting recognition engine
IRPT_ICON	Icon input	Icon active area identifier and status
IRPT_INPUT_STATUS	Pen input with pen status	Pen status information
IRPT_KEY	Soft keyboard input	Keycode, coordinate of pen touch
IRPT_NONE	No interrupt	N/A
IRPT_PEN	Pen input	Pointer to (x,y) list
IRPT_RTC	System clock alarm	N/A
IRPT_TIMER	Timer timeout and alarm	N/A
IRPT_UART	UART data transfer	Data transmission status
IRPT_USER	User	User defined

15.1.1 IRPT_AUDIO

It is a message generated from audio tools. An audio stops after it has finished or the user has called AudioStopTone() or AudioStopWave(). After audio playing stops, this interrupt is sent to the task that called AudioPlayTone() or AudioPlayWave() to indicate that the audio playing is finished.

15.1.2 IRPT_PEN

It is a message generated from a pen active area. When the application defines an active area as pen area on the display, this message is sent to the application when pen input sequence occurs over this active area. The message returns the coordinates of the pen input points.

The data message returned by IRPT_PEN consists of a list of 16-bit words. Each pair of 16-bit words in the list represents the x and y coordinate of a pen input point on the touch panel. There will always be at least 1 pair of coordinate. A pair of (-1,-1) signals the end of the list.

15.1.3 IRPT_INPUT_STATUS

This message is sent to the application to report the pen action status at the beginning and the end of each pen action within a valid pen active area. For example, when an active area created for pen input is touched, this message, together with the pen status, is sent to the application before any of the pen (x, y) coordinates are sent; then, when the pen stroke is finished and the pen leaves the touch panel, this message is again sent to the application after the last (x, y) coordinate to report the pen-up condition. This status allows the application to be aware of the pen action sequence, whether it has been dragged in from another area, or it is a pen down, etc. *Table 15-2* shows the messages types that are returned. *Figure 15-2* shows the pen actions that would generate these pen

Table 15-2 Messages generated for IRPT_INPUT_STATUS

Message	Description
PPSM_INPUT_TOUCH	A pen down condition
PPSM_INPUT_DRAG	Pen is dragged in from another area
PPSM_INPUT_PEN_UP	Pen has left the touch panel
PPSM_INPUT_DRAG_UP	Pen is dragged out of the current pen area into another area

messages.

15.1.4 IRPT_ICON

Icon area is for the purpose of selection only. It does not yield any coordinate data from the pen interrupt handler. When an icon active area is touched, PPSM sends a soft interrupt with its identifier to the application□ interrupt buffer. Two soft interrupts will be generated from each action: one interrupt for pen-down or pen drag in, and one interrupt for pen-up or pen drag up. This type of area is designed for buttons, and selection icons.

Table 15-3 shows the messages types that are returned.

Table 15-3 Messages generated for IRPT_ICON

Message	Description
PPSM_ICON_TOUCH	An icon pen down condition
PPSM_ICON_DRAG	Pen is dragged in from another area
PPSM_ICON_PEN_UP	Pen has left the touch panel
PPSM_ICON_DRAG_UP	Pen is dragged out of the current pen area into another area

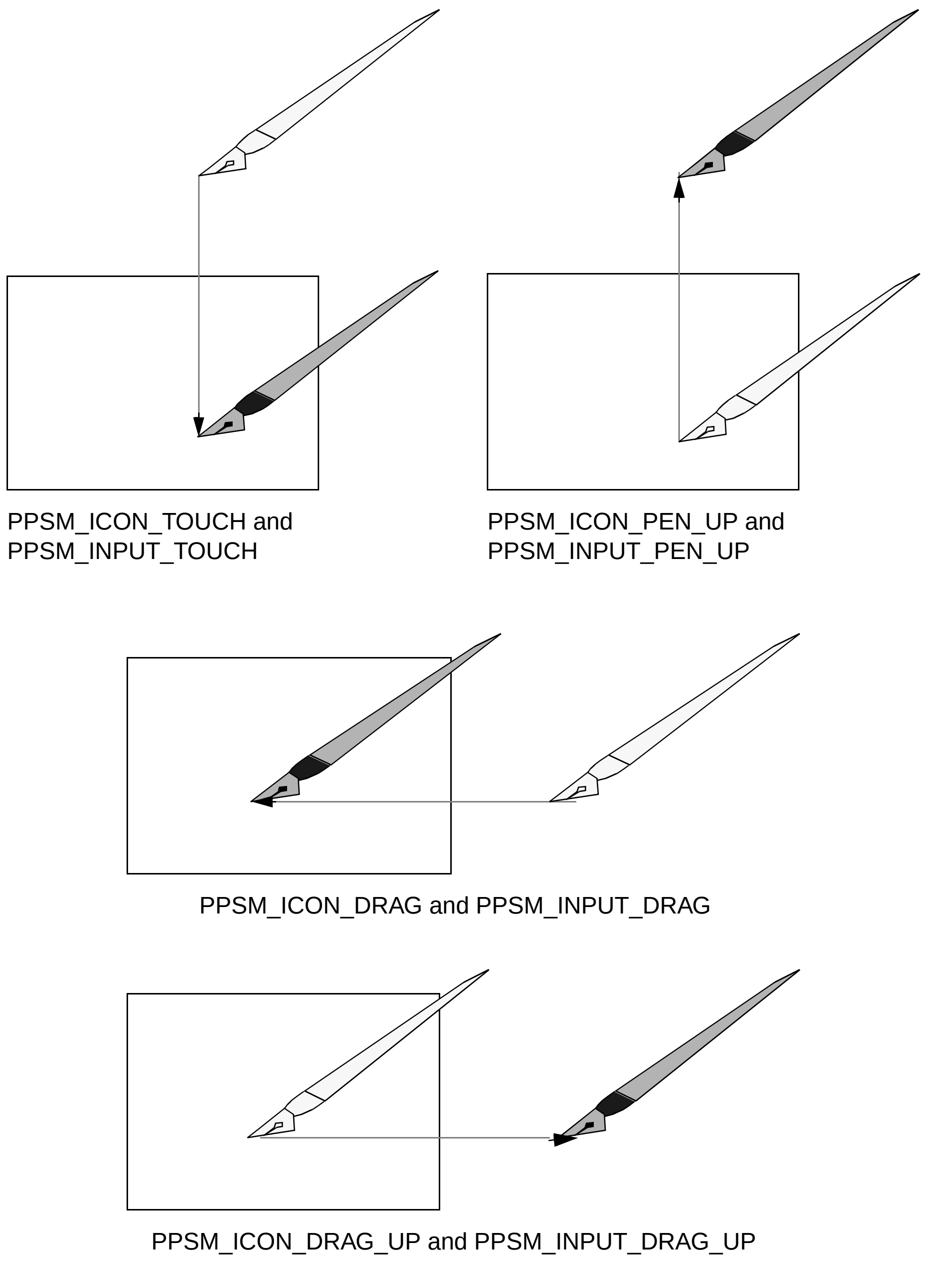

PPSM_ICON_TOUCH and
PPSM_INPUT_TOUCH
PPSM_ICON_PEN_UP and
PPSM_INPUT_PEN_UP
PPSM_ICON_DRAG and PPSM_INPUT_DRAG
PPSM_ICON_DRAG_UP and PPSM_INPUT_DRAG_UP

Figure 15-2 ICON and INPUT area pen status messages

15.1.5 IRPT_KEY

PPSM provides a soft keyboard as part of the character input tool. Once activated, a keyboard is displayed on the display area. Pressing any one of the keys on the soft keyboard will result in a IRPT_KEY message generated by PPSM to the application. The message will also include the ASCII code of the key that was pressed. The ASCII code returned is of type TEXT, i.e. 2-byte format with zero extended in high byte and the coordinate of pen touch on the key.

15.1.6 IRPT_RTC

This is the clock alarm. When an application sets an alarm for a specific time, this message will be generated by PPSM to the application when the time is reached. No data is included in this message.

15.1.7 IRPT_TIMER

Time-out message. This message is sent to the application when a specified time-out period is reached. No data is included in this message.

15.1.8 IRPT_HWR

PPSM provides an input method for handwriting character input (refer to *Chapter 5 - Character Input Methods*). If a handwriting recognition engine is used, the resultant characters generated by it are passed on to the application using this message. The data passed to the application is a list of language codes of the character candidates and the size of this list in number of bytes.

15.1.9 IRPT_UART

When an application is granted permission to use the UART (see *Chapter 12 - UART Communication Support*), this message is generated to the application to report UART data transmission status.

A 16-bit message data is also sent to the application with this message. It can have one of the following values:

Table 15-4 Messages generated for IRPT_UART

Message	Description
UART_DATA_SENT	Data sent request has been completed
UART_DATA_RECEIVED	Data has been received from the UART
UART_ERROR	An UART error condition has occurred. ` UART_ERROR An additional 16-bit word follows that identifies the error condition: ` UART_ERR_TMOUT ` UART_ERR_FRAME ` UART_ERR_PARITY ` UART_ERR_OVERRUN ` UART_ERR_NODATA

15.2 Device Interrupts

PPSM supports another set of interrupt identifiers and handlers for hardware devices. The list of device interrupt identifiers are list in *Table 15-5*.

Table 15-5 Interrupt Identifiers and User Defined Handlers

Interrupt Source	Interrupt Identifier	PPSM User Defined Handler
SPI Master	IRPT_SPIM	_SPIMIrptHandler
SPI Slave	IRPT_SPIS	_SPISIrptHandler(DragonBall only)
IRQ1	IRPT_IRQ1	_IRQ1IrptHandler
IRQ2	IRPT_IRQ2	_IRQ2IrptHandler
IRQ3	IRPT_IRQ3	_IRQ3IrptHandler
IRQ6	IRPT_IRQ6	_IRQ6IrptHandler
INT0 - INT7	IRPT_INT	_INTIrptHandler
WatchDog	IRPT_WDOG	_WatchdogIrptHandler
PWM	IRPT_PWM	_PWMIrptHandler
UART	IRPT_UART	_UARTIrptHandler
User Defined	IRPT_USER	None

15.2.1 User Defined Interrupt Handlers

For each of the external interrupts not used by PPSM system, user can install their own handler. The generic, or stub, handler source is provided in the PPSM device library. These are:

` SPI Master
` SPI Slave(DragonBall only)
` IRQ6
` IRQ3
` IRQ2
` IRQ1
` INT0 - INT7
` Watch Dog
` PWM
` UART

Each of the stub handler is associated with an external interrupt. For example, _IRQ6IrptHandler is associated with the IRQ6 external interrupt. When the external interrupt event occurs, PPSM automatically calls up the associated handler as part of the interrupt handling procedure.

For system that uses any of the external interrupts, they can supply their own handler such that integration into PPSM is possible.

For PPSM source licensee, if _UARTIrptHandler() needs to be used, "-DNO_UART_HANDLER" needs to be included in the compiler option to indicate that the internal PPSM UART interrupt handler need not be used.

15.2.2 Device Interrupt Identifiers

The device interrupt identifiers listed in *Table 15-5* can be used by the interrupt handlers to send soft interrupt messages from the handler to the application, much like the system pen and timer interrupt identifiers. PPSM provides a tool, IrptSendData(), to allow messages be sent from the user installed interrupt handlers to the application.

For example, user installed IRQ6 handlers can use IrptSendData() to send a message to the application from the IRQ6 handler to inform the application of the event, or to pass data from the hardware layer to the application layer.

15.2.3 Application Access to Handlers

By isolating the interrupt handler from the application, multiple applications can have access to the same hardware resource. However, an interrupt handler can only be registered with a single application at any one time. If more than one application is requesting the services of a single handler, it will be granted to the first application making the request. The other applications cannot access the handler until it is released by the first application.

To control access conflict between multiple application tasks accessing the same hardware concurrently, a set of interrupt tools are defined.

` IrptRequest()	Requests for an interrupt handler
` IrptRelease()	Releases an interrupt handler

15.2.4 Request and Release Interrupt Handler Service

U16 **IrptRequest**(U16 *handlerFlag*)

U16 **IrptRelease**(U16 *handlerFlag*)

When an application task needs the resource of a particular hardware peripheral, it must first request for service with the interrupt handler. Once the task successfully registers with the handler, all messages received from that peripheral are directed to the registered task until the task releases the service of the handler.

One application task can request and register with any number of interrupt handlers in the system, but each of the handlers in the system can only be attached to one single application task. Because of this, applications tasks must release the handler after use in order for the peripherals to be fully utilized.

To request or release a handler, the flags in the *Table 15-6* below are used in all the interrupt tools to specify which of the handlers to use. These flags are bit values representing individual interrupt handlers.

Table 15-6 Interrupt Handler Flags

Interrupt Handler	Interrupt Flag
SPI Master	IRPT_SPIM_FLAG

Table 15-6 Interrupt Handler Flags

Interrupt Handler	Interrupt Flag
SPI Slave	IRPT_SPIS_FLAG(DragonBall only)
IRQ1	IRPT_IRQ1_FLAG
IRQ2	IRPT_IRQ2_FLAG
IRQ3	IRPT_IRQ3_FLAG
IRQ6	IRPT_IRQ6_FLAG
INT0 - INT7	IRPT_INT_FLAG
WatchDog	IRPT_WDOG_FLAG
PWM	IRPT_PWM_FLAG
UART	IRPT_UART_FLAG
User Defined	IRPT_USER_FLAG

The IRPT_USER is intended for internal software interrupt use. For example, user defined interrupt messages to inform caller of particular event. Users can specify their own meanings to this message.

Example 15-1 Request an Interrupt Handler

```
/* request usage of UART */
if ( IrptRequest(IRPT_UART_FLAG) != IRPT_UART_FLAG )
  return (PPSM_ERROR);
.
.
.
/* Release the handler after use */
IrptRelease(IRPT_UART_FLAG);
```

15.3 Message Handling

STATUS **IrptSendData**(U16 *irptType,* U32 *sData,* P_U32 *data,* U32 *size*)

U16 **IrptGetData**(P_U32 *sData*, P_U32 **data*, P_U32 *size*)

User can send messages, with or without extra data information, to the application using IrptSendData(). This tool can only be used within a user installed handler.

When an application requested and has been granted access to a user installed handler, any messages sent by IrptSendData() from that handler will always be directed to that application, until the application has released the handler.

IrptSendData() appends the message from the handler at the end of the application□ software interrupt buffer. The receiving application retrieves the message via the system tool IrptGetData(), in the same manner as other system messages.

15.3.1 Example

Assuming that an application has already given access to an IRQ6 handler by PPSM, when the user defined IRQ6 handler calls IrptSendData(), the data are sent to the application immediately.

```
shortMessage = COMMAND_EVENT;
messageSize = 4;
messageData = pInData;
IrptSendData( IRPT_IRQ6, shortMessage, &messageData, messageSize);
```

The application retrieve the message via the IrptGetdata() call.

```
switch (IrptGetData( &event, &InData, &InSize))
.
.
.
case IRPT_IRQ6:
      /*  IRQ6 event has occurred */
      if (event == COMMAND_EVENT)
      .
      .
      .
      default;
```

In the above example, the following data are passed from the IRQ6 handler to the application:

Table 15-7 Data Passed from Handler to Application

IrptSendData		**IrptGetData**	
Argument	**Data**	**Argument**	**Data**
irptType	IRPT_IRQ6	return value	IRPT_IRQ6
sData	*shortMessage*	*sData*	*event*
data	*messageData*	*data*	*InData*
size	*messageSize*	*size*	*InSize*

Chapter 16 Using System Tools

PPSM provides additional tools for applications to access and control system resources.

16.1 PPSM Initialization

STATUS **PPSMInit**(U16 *calibration*)

PPSM needs to be initialized. PPSMInit() must first be called before any of the PPSM tools can be called. It performs all the system initialization, hardware devices initialization and performs calibration for the pen input panel and the LCD display.

A device driver (PenInit.c) function CalibratePen(U16 logoFlag) is called to set the touch panel origin and touch panel maximum value if PPSMInit(TRUE) is called. This function should call PenSetInputOrg(X, Y) and PenSetInputMax(X, Y). The coordinate of origin is the top left corner of the touch panel in terms of display screen coordinate. The coordinate of maximum is the bottom right corner of the touch panel in terms of display screen coordinate.

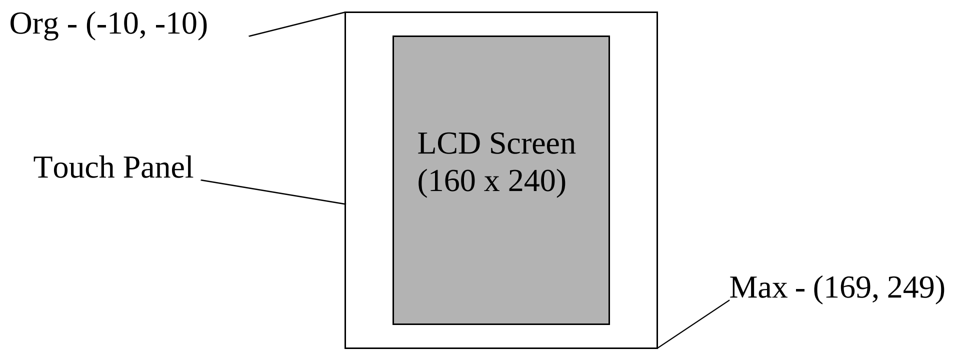


Figure 16-1 example of calibarting a system

The default touch panel calculation value is based on a touch panel that has 320 pixels by 240 pixels (physical sizes 12 cm by 9 cm), using a 10-bit A/D convertor. An offset of 100 (in A/D output unit) is chosen as the A/D non-linear area around the edge of the touch panel.

The caller of CalibratePen() can specify if logo display is required or not. If the logoFlag is FALSE, no logo is displayed. By default, a Motorola Logo and two cross-hairs (one at top right corner and the other at bottom left corner) are displayed on the LCD screen for calibration. The user must press these two cross-hairs, in no particular order, before PPSM will continue execution.

Example 16-1 Initialize PPSM

```
/* Initialize PPSM with pen calibration */
PPSMInit(TRUE);
```

16.1.1 Motorola Logo

Depending on the physical LCD size, an appropriate Motorola logo is displayed on the LCD display during pen calibration stage.

For LCD display that is larger than 280 pixels wide by 150 pixels high, the standard logo will be displayed. This logo is 256 pixels wide by 97 pixels high, *Figure 16-2*.

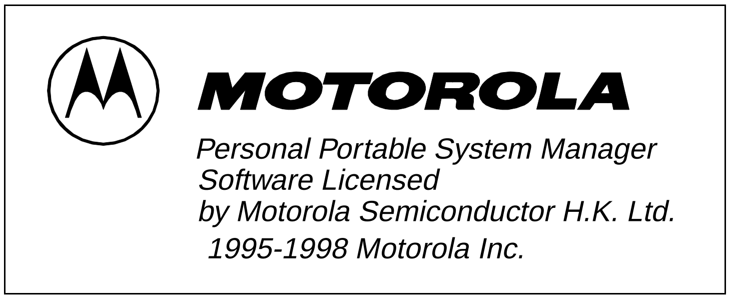

MOTOROLA
Personal Portable System Manager
Software Licensed
by Motorola Semiconductor H.K. Ltd.
1995-1998 Motorola Inc.

Figure 16-2 Standard Motorola logo

For LCD display that is larger than 150 pixels wide by 80 pixels high, but smaller than 280 pixels by 150 pixels, a smaller Motorola logo will be used. This smaller Motorola logo will be 104 pixels by 25 pixels, *Figure 16-3*.

Figure 16-3 Small Motorola logo

For LCD display that is smaller than 150 pixels wide by 80 pixels high, no Motorola logo will be drawn.

Chapter 17 Audio Tools

17.1 Audio Playing

PPSM supports two types of audio, wave playing and tone playing. Due to the hardware limitation, the wave playing is only available for DragonBall-EZ.

The audio tools have the following properties.

- Only one wave file or tone can be played during a given moment.
- A wave file or tone cannot be played if the PWM(Pulse Width Modulation) module is in used by another task or application
- An interrupt "IRPT_AUDIO" message will be sent to the task that called AudioPlayTone() or AudioPlayWave() to indicate the audio playing has finished.

17.2 Tone playing

STATUS **AudioPlayTone**(P_U16 toneData, U32 toneSize, U16 toneDuration, U8 autoRepeat)

PPSM supports tone playing for both DragonBall and DragonBall-EZ through the PWM module. Tone playing can play a melody with user specified fixed duration and changeable frequencies throughout that duration. For better frequency resolution, the tone frequency is limited between 31Hz and 4048Hz.

Name	Description
toneData	The pointer to the tone sequence, with frequencies between 31Hz and 4048Hz.
toneSize	Total number of tone frequencies to be played.
toneDuration	The duration of each tone frequency For DragonBall-EZ - TONE_DUR_512Hz - TONE_DUR_256Hz - TONE_DUR_128Hz - TONE_DUR_64Hz - TONE_DUR_32Hz - TONE_DUR_16Hz - TONE_DUR_8Hz - TONE_DUR_4Hz For DragonBall 0 to 1000, length of duration in number of milliseconds.

Name	Description
autoRepeat	To indicate if auto-repeat is needed or not 0 - no autorepeat. 1 - autorepeat.

Example 16-1 PPSM tone playing

```
/*  100Hz, 1000Hz, 500Hz and 600Hz */
U16  toneData[] = {100, 1000, 500, 600};

/*  Play a melody with 4 different tone frequencies, each with 250ms duration */
#ifdef EZ328
    AudioPlayTone((P_U16)toneData, 4, TONE_DUR_4HZ, 1);
#else
    AudioPlayTone((P_U16)toneData, 4, 250, 1);
#endif
```

To stop the tone playing, a user can call AudioStopTone(). To check if the Audio Tools are currently being used, a user can call AudioInUse().

Note: This is impossible to play a tone with value of frequency less than the value of duration, since the duration of this frequency is longer than the allowed duration.

17.3 Wave playing (DragonBall-EZ only)

PPSM audio tools can play back a PCM(Pulse Code Modulation) audio wave file that can be generated by many audio programs. Wave playing can be done by two PPSM audio tools - AdvAudioPlayWave() and AudioPlayWave().

AdvAudioPlayWave() is provided for users with solid knowledge of PWM who want to have advanced configuration details over the DragonBall-EZ PWM module. For most cases, AudioPlayWave should be used.

STATUS **AudioPlayWave**(P_U8 waveData, U32 waveSize, U8 samplingRate)

Name	Description
waveData	The pointer to the PCM audio wave signal
waveSize	Total number of data bytes occupied by the audio signal
samplingRate	The requested sampling rate ` SAMPLING_32KHZ ` SAMPLING_16KHZ ` SAMPLING_11KHZ ` SAMPLING_8KHZ ` SAMPLING_4KHZ

Example 16-2 PPSM wave playing

```
/*  Some PWM wave data */
U16  waveData[] = {...};
```

```
/*  Play a melody with 1000 data bytes at 16kHz sampling/reconstruction rate */
AudioPlayWave((P_U8)waveData, 1000, SAMPLING_16KHZ);
```

STATUS **AdvAudioPlayWave**(P_U8 waveData, U32 waveSize, U8 prescaler, U8 repeat, U8 clksel)

Name	Description
waveData	The pointer to the PCM audio wave signal
waveSize	Total number of byte of the audio signal
prescaler(see DragonBall-EZ user□ manual)	Bit 14~8 of the PWM control register, value from 0 to 127
repeat(see DragonBall-EZ user□ manual)	Bit 2,3 of the PWM control register, value from 0 to 3
clksel(see DragonBall-EZ user□ manual)	Bit 0,1 of the PWM control register, value from 0 to 3

The sampling rate can be calculated by the above input parameters.

$$SamplingRate = ((16.58MHz)\ \ (prescalar + 1)\ \ (clksel)\ \ (repeat)\ \ 256)$$

For more detail information, please refer to the DragonBall-EZ user□ manual.

Example 16-3 PPSM wave playing

```
/*  Some PWM wave data */
U16 waveData[] = {...};

/*  Play a melody with 1000 data bytes at 16kHz sampling/reconstruction rate */
/*  16kHz = 16.58Mz/(2 x 1 x 2 x 256) */
AdvAudioPlayWave((P_U8)waveData, 1000, 0, 2, 0);
```

The device driver function _PWMIrptHandler() under IrptDev.c should return TRUE for proper wave playing with PPSM audio tools. If the user is going to use his/her own PWM interrupt function and wants to disable PPSM audio wave playing tools, the _PWMIrptHandler() should return FALSE instead.

17.4 Stop the audio playing

An audio stops after it has finished or the user has called AudioStopTone() or AudioStopWave(). After audio playing stops, an interrupt is sent to the task that called AudioPlayTone() or AudioPlayWave() to indicate that the audio playing is finished.

Example 16-4 Stopping an audio play

```
/*  Some PWM wave data */
U16 waveData[] = {...};

/*  Play a melody with 1000 data bytes at 16kHz sampling/reconstruction rate */
AudioPlayWave((P_U8)waveData, 1000, SAMPLING_16KHZ);

switch( IrptGetData((P_U32)&id, (P_U32*) &inData, (P_U32) &size) )
{
case IRPT_AUDIO:
    /*  Display message to indicate the audio has stopped */
    ....
    break;
case IRPT_ICON:
    /*  Click icon to stop the wave playing */
    rv = AudioStopWave();
    ....
    break;
}
```

Part III
API Toolset

Chapter 18 Pen Input Tools

18.1 ActiveAreaDisable

Syntax

STATUS **ActiveAreaDisable**(U32 *areaId*)

Description

Removes an active area entry from the active scan list. The identifier *areaId* specifies the entry that is to be deleted. This identifier must be a valid identifier obtained from the ActiveAreaEnable().

Parameter

Name	Description
areaId	Identifier of the active area to be removed from the active area list

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_AREA_ID	Invalid active area identifier

18.2 ActiveAreaEnable

Syntax

STATUS **ActiveAreaEnable**(P _U32 *areaId*, U32 *code*, U32 *mode*, S16 *xSrc*,
S16 *ySrc*, S16 *xDest*, S16 *yDest*)

Description

Creates and enables an active area onto the application□ list of active input area. An active area is defined as a rectangular region of the display panel where a software interrupt message will be sent to the application that created the area when the region is pressed. For the type of messages that are returned to the application, see the tool IrptGetdata().

Active areas can be created on or off the LCD display area but must be within the boundary of the touch panel. The properties of the areas on and off the display area are the same except for those areas that are outside the LCD display area, echoing is NOT allowed.

Parameter

Name	Description
areaId	Returns an active area identifier. This identifier is used by the PPSM to refer to the active area until it is removed from the list.
code	Type of active area. It takes either one of the following two value: ` ICON_AREA Area for icon ` INPUT_AREA Area for pen input
mode	This argument is valid only if INPUT_AREA is selected. It can take one of the following modes: ` STROKE_MODE One interrupt per input stroke ` CONTINUOUS_MODE One interrupt per sampled points ` CONFINED_MODE same as STROKE_MODE but pen confined within active area
xSrc	Top left x-coordinate of the active area
ySrc	Top left y-coordinate of the active area
xDest	Bottom right x-coordinate of the active area
yDest	Bottom right y-coordinate of the active area

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_AREA_ID	Invalid active area identifier pointer
PPSM_ERR_AREA_CODE	Invalid area code
PPSM_ERR_COORDINATE	Invalid coordinates
PPSM_ERR_NO_MEMORY	Not enough memory

18.3 ActiveAreaRead

Syntax

STATUS **ActiveAreaRead**(U32 *areaId*, P_S16 *xSrc*, P_S16 *ySrc*, P_S16 *xDest*, P_S16 *yDest*)

Description

Reads the area coordinates of an active area entry in the active scan list. The identifier *areaId* specifies the entry that is to be read.

Parameter

Name	Description
areaId	Active area identifier
xSrc	Returns the top left x-coordinate of the active area
ySrc	Returns the top left y-coordinate of the active area
xDest	Returns the bottom right x-coordinate of the active area
yDest	Returns the bottom right y-coordinate of the active area

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_AREA_ID	Invalid active area identifier

18.4 ActiveAreaSuspend

Syntax

STATUS **ActiveAreaSuspend**(U32 *areaId*, U32 *flag)*

Description

Suspend or re-enable an active area that has already been created. When an active area is suspended, it will no longer response to pen-input interrupt but it will remain in the active list. When an active area is re-enabled, it will once again respond to pen-input.

Parameter

Name	Description
areaId	Active area identifier, must be a valid identifier returned by PPSM.

Name	Description
flag	Flag to indicate whether to suspend or re-enable the active area: ` AREA_SUSPEND Suspend the active area ` AREA_REENABLE Re-enable the active area

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_AREA_ID	Invalid active area identifier

18.5 ActiveAreaToFront

Syntax

STATUS **ActiveAreaToFront**(U32 *areaId*)

Description

Given the active area identifier, this tool will extract the element from the active area linked list and insert the element at the front of the list.

Once the element is at the front of the list, it becomes the active area to receive pen input if other active areas that are overlapping the same physical area.

Parameter

Name	Description
areaId	Active area identifier. Must be a valid identifier returned by PPSM

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_AREA_ID	Invalid active area identifier

18.6 ActiveListPop

Syntax

STATUS **ActiveListPop**(void)

Description

Pops the top background active list of the current task from the active area stack. The active list that is currently being used is destroyed, replaced by the top background active list.

Parameter

Name	Description
None	

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERROR	Active list stack empty

18.7 ActiveListPush

Syntax

STATUS **ActiveListPush**(void)

Description

Pushes the current active area list of the current task into background and creates a new empty active area list. Once in the background, scanning on these areas is disabled. Any new active areas registering to PPSM will belong to the new active list.

The number of active lists in the background is restricted to 8 levels. They are stored internally in an active area stack.

Parameter

Name	Description
None	

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_ACTIVE_PUSH	Unable to push active list

18.8 AreaEchoDisable

Syntax

STATUS **AreaEchoDisable**(U32 *areaId*)

Description

Disables active area pixel echoing mode. Once disabled, all pen-input device selected pixels will not be echoed back on the LCD display. Echoing is disabled for all active input area by default.

Parameter

Name	Description
areaId	Active area identifier

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_AREA_ID	Invalid active area identifier

18.9 AreaEchoEnable

Syntax

STATUS **AreaEchoEnable**(U32 *areaId*)

Description

Enables active area pixel echoing mode. Once enabled, all pen-input device selected pixels will be echoed back on the LCD display. Echoing is disabled for all active input area by default.

Parameter

Name	Description
areaId	Active area identifier

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_AREA_ID	Invalid active area identifier

18.10 ActiveAreaPosition

Syntax

STATUS **ActiveAreaPosition**(U32 *areaId*, S16 *xSrc*, S16 *ySrc*, S16 *xDest*, S16 *yDest*)

Description

This function will change the position and the size of the active area specified by *areaId*.

Parameter

Name	Description
areaId	Existing valid active area id.
xSrc	Top left corner x coordinate
ySrc	Top left corner y coordinate
xDest	Bottom right x coordinate
yDest	Bottom right y coordinate

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_AREA_ID	Error if area id is invalid
PPSM_ERR_COORDINATE	Error if any point is outside touch panel or the active area lies in the boundary between LCD area and touch panel only area.

18.11 CtrlIconDisable

Syntax

STATUS **CtrlIconDisable**(U32 *iconId*)

Description

Removes a predefined icon area from an application. The argument *iconId* must be a valid active area identifier supplied by PPSM.

Parameter

Name	Description
iconId	The identifier of the predefined icon area to be removed

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_AREA_ID	Invalid active area identifier

18.12 CtrlIconEnable

Syntax

STATUS **CtrlIconEnable**(P_U32 *iconId*, S16 *xSrc,* S16 *ySrc,* U16 *iconType*)

Description

Adds a predefined icon area to an application. The icon area can be placed anywhere on the application□ display area. The argument (*xSrc*, *ySrc*) specifies the position of the icon□ top left corner. An area identifier is returned to the caller.

Parameter

Name	Description
iconId	Returns an area identifier. This identifier is used by the PPSM to refer to the predefined icon area until it is removed from the list.
xSrc	X coordinate for the predefined icon□ top left corner
ySrc	Y coordinate for the predefined icon□ top left corner

Name	Description
iconType	The type of predefined icon to use. It can be any of the followings: ` PPSM_ICON_8_UP 8x8 Up Icon ` PPSM_ICON_8_DOWN 8x8 Down Icon ` PPSM_ICON_8_LEFT 8x8 Left Icon ` PPSM_ICON_8_RIGHT 8x8 Right Icon ` PPSM_ICON_8_DONE 8x16 Done Icon ` PPSM_ICON_16_UP 16x16 Up Icon ` PPSM_ICON_16_DOWN 16x16 Down Icon ` PPSM_ICON_16_LEFT 16x16 Left Icon ` PPSM_ICON_16_RIGHT 16x16 Right Icon ` PPSM_ICON_16_DONE 16x32 Done Icon

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_ICON_TYPE	Invalid predefined icon type
PPSM_ERR_COORDINATE	Invalid coordinates
PPSM_ERR_AREA_ID	Invalid active area identifier pointer

18.13 IconScanOff

Syntax

void **IconScanOff**(void)

Description

Switches off system application icon scanning of all icon active areas of the current task.

Parameter

Name	Description
None	

Return Value

Name	Description
None	

18.14 IconScanOn

Syntax

void **IconScanOn**(void)

Description

Switches on system application icon scanning for the current task. This is on by default.

Parameter

Name	Description
None	

Return Value

Name	Description
None	

18.15 PenCalibration

Syntax

STATUS **PenCalibration**(U16 logoFlag)

Description

This function performs pen calibration routine. It calls CalibratePen()(in peninit.c of the device driver library) to calibrate the touch panel. User may use different calibration method by changing CalibratePen() in peninit.c. The default driver will clear the screen and wait for pen data until 2 valid points for pen calibration are captured.

Parameter

Name	Description
logoFlag	TRUE - Put a Motorola logo on screen FALSE - No Motorola logo will be put.

Return Value

Name	Description
PPSM_OK	Successful operation

18.16 PenEchoParam

Syntax

STATUS **PenEchoParam**(U16 *echoCol,* U16 *echoWidth*)

Description

This tool allows the application to set the color and the drawing width for system pen echoing.

Parameter

Name	Description
echoCol	Pen echo color. For 1bit/pixel graphics, can be: ` WHITE ` BLACK. For 2bit/pixel graphics, can be: ` WHITE ` LIGHT_GREY ` DARK_GREY ` BLACK
echoWidth	Pen echo width in number of pixels

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_COLOUR	Invalid pen color
PPSM_ERROR	Unsuccessful operation

18.17 PenGetInput

Syntax

STATUS **PenGetInput**(P_S16 *xPos*, P_S16 *yPos*)

Description

Returns a single pair of X and Y coordinates of the pen-touch panel contact point. A set of -1 will be returned from this module if the pen is out of the touch panel range, i.e. pen up.

Parameter

Name	Description
xPos	Returns the X position of pen input point
yPos	Returns the Y position of pen input point

Return Value

Name	Description
PPSM_OK	Successful operation

18.18 PenSetInputMax

Syntax

STATUS PenSetInputMax(S16 x, S16 y)

Description

It allows the user to define the bottom-right corner of the touch panel coordinate in terms of screen display coordinate. It is usually called when the system is doing calibration/re-calibration and should only be called by *CalibratePen(U16 logoFlag)* at the device driver level.

Parameter

Name	Description
x	x-coordinate of the bottom-right corner of the touch panel coordinate in turn of screen display coordinate.
y	y-coordinate of the bottom-right corner of the touch panel coordinate in turn of screen display coordinate.

Return Value

Name	Description
PPSM_OK	Successful operation

18.19 PenSetInputOrg

Syntax

STATUS **PenSetInputOrg**(S16 x, S16 y)

Description

It allows the user to define the top-left corner of the touch panel coordinate in terms of screen display coordinate. It is usually called when the system is doing calibration/re-calibration and should only be called by *CalibratePen(U16 logoFlag)* at the device driver level.

Parameter

Name	Description
x	x-coordinate of the top-left corner of the touch panel coordinate in turn of screen display coordinate.
y	y-coordinate of the top-left corner of the touch panel coordinate in turn of screen display coordinate.

Return Value

Name	Description
PPSM_OK	Successful operation

18.20 PenSetRate

Syntax

STATUS **PenSetRate**(U16 *samplingPeriod*)

Description

To allow user to define the period for pen sampling dynamically. The sampling period set in an task does not affect other task`s sampling period. It only takes effect after at least one active area has been created under the calling task.

Parameter

Name	Description
samplingPeriod	Sampling period in milliseconds. This is the time between A/D sampling of the pen input coordinates. It has a range of 1 to 1000 milliseconds (e.g. *samplingPeriod* of 25milliseconds translates to 40 samples/ sec)

For MC68EZ328 only

Sampling Period	Pen Sampling Rate
samplingPeriod>= 250	4Hz
250>samplingPeriod>=125	8Hz
125>samplingPeriod>= 62	16Hz
62>samplingPeriod>= 31	32Hz
31>samplingPeriod>= 15	64Hz
15>samplingPeriod>= 7	128Hz
7>samplingPeriod>= 3	256Hz
3>samplingPeriod>= 1	512Hz

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_PEN_RATE	Invalid sampling period specified

18.21 ScanningOff

Syntax

void **ScanningOff**(void)

Description

Switches off touch panel scanning for the current application. All application active areas will not response to pen-input interrupt.

Parameter

Name	Description
None	

Return Value

Name	Description
None	

18.22 ScanningOn

Syntax

void **ScanningOn**(void)

Description

Switches on touch panel scanning for the current application. This will enable all application active area scanning of the current task.

Parameter

Name	Description
None	

Return Value

Name	Description
None	

Chapter 19 Character Input Tools

19.1 AdvOpenInputPad

Syntax

STATUS **AdvOpenInputPad**(U16 xPos, U16 yPos, U16 numRow, U16 numCol, U16 areaWidth, U16 areaHeight, U16 echoCol, U16 echoWidth, U32 timeOut, U16 samplingTime, U8 areaClean, U16 stackSize)

Description

Opens the input pad for handwritten character input(similar to the tool OpenInputPad() but with advanced configuration details). It allows the caller to specify:

` position of the input pad
` number of rows and columns of input boxes
` the width and the height of the input boxes
` the echo ink colour and dot width
` the length of time out after a stroke
` the sampling rate of the pen
` if the system should clean the input box for the user after each character is written
` the stack size for the input pad subtask

Parameter

Name	Description
xPos	X-coordinate of the top left corner of the input pad
yPos	Y-coordinate of the top left corner of the input pad
numRow	Number of rows of input boxes
numCol	Number of columns of input boxes
areaWidth	Width of each input box in number of pixels
areaHeight	Height of each input box in number of pixels
echoCol	Colour of the echo ink
echoWidth	Dot width of the echo ink

Name	Description
timeOut	Length of time to wait between a written stroke and recognition start in number of milliseconds with range of 0 to 1000. If it is zero, time-out is disabled. So, recognition will only start after writing a stroke in another input box.
samplingTime	It is the time between two pen samples. It has range of 0 to 1000 milliseconds
areaClean	cleans after each character input or not 0 - do not clean 1 - clean
stackSize	the stack size of input pad subtask

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_INPUT_PAD_OPENED	Input pad is already opened - by the same task or its sibling sub-task or its parent task.
PPSM_ERR_INPUT_PAD_WIDTH	Invalid width
PPSM_ERR_INPUT_PAD_HEIGHT	Invalid height
PPSM_ERR_INPUT_PAD_X_POS	Invalid X-coordinate
PPSM_ERR_INPUT_PAD_Y_POS	Invalid Y-coordinate
PPSM_ERR_PEN_RATE	Invalid pen sampling time
PPSM_ERR_PAN_INIT	Panning screen has not been initialized
PPSM_ERR_NO_MEMORY	Not enough memory
PPSM_ERR_TMOUT_VALUE	Invalid time out value

19.2 AdvOpenSoftKey

Syntax

STATUS **AdvOpenSoftKey**(U16 xPos, U16 yPos, U16 keyWidth, U16 keyHeight, U16 numCol, U16 numRow, P_U16 keyMap, P_U8 bitmap)

Description

Opens a soft keyboard module in a similar manner as the tool OpenSoftKey() but with advanced configuration details. It allows the caller to specify:

` location of the soft keyboard

- width and height of each key in number of pixels
- number of rows and columns of keys
- the return code of each key(ie, key code or scan code)
- the bitmap of the soft keyboard user interface

Parameter

Name	Description
xPos	X-coordinate of the top left corner of the soft keyboard
yPos	Y-coordinate of the top left corner of the soft keyboard
keyWidth	Width of each key in number of pixels
keyHeight	Height of each key in number of pixels
numRow	Number of rows of keys
numCol	Number of columns of keys
keyMap	Pointer to an array of return key codes, from the top left key towards to the right, then next row and so on, until the bottom right key

bitmap	bitmap of the soft keyboard area interface
	width = (keyWidth * numCol)
	height = (keyHeight * numRow)
	in number of pixels

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_SKBD_USED	Soft keyboard has already being used by current task
PPSM_ERR_PAN_INIT	Panning screen info is not initialized yet
PPSM_ERR_INPUT_PAD_NOSCREEN	No panning screen memory is allocated for this task
PPSM_ERR_SKBD_XSIZE	Soft keyboard x-coordinate out of range
PPSM_ERR_SKBD_YSIZE	Soft keyboard y-coordinate out of range
PPSM_ERR_NO_MEMORY	Not enough memory

19.3 CloseInputPad

Syntax

STATUS **CloseInputPad**(void)

Description

Closes the handwritten character input pad.

The input pad image is removed from the panning screen display and no more handwriting recognition messages (IRPT_HWR) will be generated from the system to the application. The original image covered by the input pad is restored by the system.

Parameter

Name	Description
None	

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_PEN_INIT	No active area found
PPSM_ERR_INPUT_PAD_CLOSED	Input pad is not opened

19.4 CloseSoftKey

Syntax

STATUS **CloseSoftKey**(void)

Description

Closes the soft keyboard module.

The soft keyboard image is removed from the display area and no more key pressed messages (IRPT_KEY) will be generated from the system to the application. The original image covered by the keyboard is restored by the system.

Parameter

Name	Description
None	

Return Value

Name	Description
PPSM_OK	Successful operation

Name	Description
PPSM_ERROR	Soft keyboard is not opened

19.5 OpenInputPad

Syntax

STATUS **OpenInputPad**(U16 *xPos*, U16 *yPos*, U16 *numRow*, U16 *numCol*,
U16 *areaSize*)

Description

Opens the input pad for handwritten character input.

The input pad is drawn at the specified position in a row by column format. The input pad has *numRow* by *numCol* number of square input boxes. Each input box is of size *areaSize* by *areaSize* pixels.

Once the input pad is opened, handwriting recognition interrupt messages (IRPT_HWR) are generated to the application when characters are being recognized. Each recognized character generates an individual interrupt message.

Only one input pad is allowed among all applications. The image that is covered by the input pad is saved by the system automatically, which will be restored upon closing of the input pad. Note that the image saved by the system is a snap-shot of the display screen at the time this tool is called. Any changes to this area by the application will not be recorded by the system. The input pad needs to be closed before any of the other applications can open it.

The default length of input timeout is 1sec.

Parameter

Name	Description
xPos	X-coordinate of the top left corner of the input pad
yPos	Y-coordinate of the top left corner of the input pad
numRow	Number of rows of input boxes
numCol	Number of columns of input boxes
areaSize	Size of each input box in number of pixels

Return Value

Name	Description
PPSM_OK	Successful operation

Name	Description
PPSM_ERR_INPUT_PAD_OPENED	Input pad already opened by the same task or its sibling sub-task or its parent task
PPSM_ERR_INPUT_PAD_X_POS	Input pad x-coordinate out of range
PPSM_ERR_INPUT_PAD_Y_POS	Input pad y-coordinate out of range
PPSM_ERR_INPUT_PAD_WIDTH	Input pad width out of range
PPSM_ERR_INPUT_PAD_HEIGHT	Input pad height out of range
PPSM_ERR_NO_MEMORY	Not enough memory

19.6 OpenSoftKey

Syntax

STATUS **OpenSoftKey**(U16 *xPos*, U16 *yPos*)

Description

Opens a soft keyboard module for type-written English character input.

A soft keyboard is drawn at the position specified by the application. Once the keyboard is opened, key-pressed interrupt messages (IRPT_KEY) are generated to the application when key icons on the soft keyboard module are pressed. Each individual key pressed generates an individual interrupt message.

Only one soft keyboard is allowed for each application. The image that is covered by the pseudo keyboard is saved by the system automatically, which will be restored upon closing of the keyboard. Note that the image saved by the system is a snap-shot of the display screen at the time this tool is called. Any changes to this area by the application will not be recorded by the system.

Parameter

Name	Description
xPos	X-coordinate of the top left corner of the soft keyboard
yPos	Y-coordinate of the top left corner of the soft keyboard

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_SKBD_X_POS	Input pad x-coordinate out of range

Name	Description
PPSM_ERR_SKBD_Y_POS	Input pad y-coordinate out of range
PPSM_ERR_SKBD_USED	Soft keyboard already being used
PPSM_ERR_NO_MEMORY	Not enough memory

Chapter 20 Graphics Tools

All coordinates mentioned in the following tools are Panning Screen coordinates:

- The range of valid (x, y) coordinates is from (0, 0) through (Panning Screen Width - 1, Panning Screen Height - 1).
- The range of valid width is from 1 through Panning Screen Width
- The range of valid height is from 1 through Panning Screen Height

20.1 ChangePanning

Syntax

STATUS **ChangePanning**(P_PAN_SCREEN *newPanning*, U16 *flag*,
P_PAN_SCREEN *oldPanning*)

Description

This function changes the current task panning screen address, size, etc. The panning screen width must be divisible by 8 for 2 bits/pixel and divisible by 16 for 1 bit/pixel display.

Parameter

Name	Description
newPanning	A PAN_SCREEN structure containing the properties to set to: ` P_U32 panAddress - Panning screen address ` U16 horzSize - Panning screen horizontal size ` U16 vertSize - Panning screen vertical size ` U16 displayXOrigin - x-coordinate of LCD display origin relative to panning screen ` U16 displayYOrigin - y-coordinate of LCD display origin relative to panning screen ` P_U32 displayScreenAddr - the LCD Display screen address used in hardware register Display Screen Address ` U8 regPOSR - the bit position offset used in hardware register POSR ` U16 regPSW - (Panning screen width * number of bit per pixel)/16
flag	FALSE if the old panning screen is not needed any more and TRUE if the old panning screen needs to be kept and returned in *oldPanning*
oldPanning	A PAN_SCREEN structure returned by the system containing the original settings: ` P_U32 panAddress ` U16 horzSize ` U16 vertSize ` U16 displayXOrigin ` U16 displayYOrigin ` P_U32 displayScreenAddr ` U8 regPOSR ` U16 regPSW

Return Value

Name	Description
PPSM_OK	Successful operation

Name	Description
PPSM_ERR_PAN_ADDRESS	Invalid panning screen address
PPSM_ERR_PAN_WIDTH	Invalid panning screen width
PPSM_ERR_PAN_HEIGHT	Invalid panning screen height

Hint

The panning screen address must be created by using GetScreenMem(). In some cases, a common panning screen may be shared among multiple tasks. To achieve that, the common panning screen is created by GetScreenMem(), and ChangePanning() is called at the beginning of each task.

20.2 ChangeWindow

Syntax

STATUS **ChangeWindow**(U32 *addr*, U16 *width*, U16 *height*, P_U32 *oldAddr*,
P_U16 *oldWidth*, P_U16 *oldHeight*)

Description

This tool will direct all graphics routines to the memory at *addr* so that nothing will be changed on the panning screen being displayed. The *width* must be divisible by 8 for 2 bits/pixel display and divisible by 16 for 1 bit/pixel display. The original settings will be returned to the calling application.

Parameter

Name	Description
addr	New graphics output area address
width	New graphics output area width in number of pixels
height	New graphics output area height in number of pixels
oldAddr	Old graphics output area address
oldWidth	Old graphics output area width in number of pixels
oldHeight	Old graphics output area height in number of pixels

Return Value

Name	Description
PPSM_OK	Successful operation

Name	Description
PPSM_ERR_WIDTH	Invalid graphics output area width
PPSM_ERR_HEIGHT	Invalid graphics output area height

Hint

This is used for displaying image which needs a long time to generate. The image can be plotted in other memory area by using ChangeWindow(). When the image is drawn, it can be copied to the panning screen by calling ChangeWindow() and PutRec().

20.3 ClearRec

Syntax

STATUS **ClearRec**(U16 *greyLevel*, U16 *xSrc*, U16 *ySrc*, U16 *width*, U16
height, U16 *style*)

Description

Fills the given area with grey level indicated by *greyLevel* with *style*.

Parameter

Name	Description
greyLevel	Grey level of the line
xSrc	Top left x-coordinate of the rectangular area
ySrc	Top left y-coordinate of the rectangular area
width	Width of the rectangular area in pixels
height	Height of the rectangular area in pixels
style	Output Style (AND_STYLE, OR_STYLE, EXOR_STYLE, or REPLACE_STYLE).

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_LCD_X	Invalid x-coordinate
PPSM_ERR_LCD_Y	Invalid y-coordinate
PPSM_ERR_WIDTH	Invalid width
PPSM_ERR_HEIGHT	Invalid height

Name	Description
PPSM_ERR_LCD_STYLE	Invalid style

Hint

If (*xSrc*, *ySrc*) is (5, 10), width equals 20, and height equals 10, this will fill the rectangle with top left corner at (5, 10) and bottom right corner at (24, 39) on LCD with specified *greyLevel* with *style*.

20.4 ClearScreen

Syntax

void **ClearScreen**(U16 *greyLevel*)

Description

Fills the whole panning screen with grey level indicated by *greyLevel*.

Parameter

Name	Description
greyLevel	Grey level of screen

Return Value

Name	Description
None	

20.5 CursorGetOrigin

Syntax

STATUS **CursorGetOrigin**(P_U16 *xPos*, P_U16 *yPos*)

Description

Returns the coordinate (**xPos*, **yPos*) of the display origin.

Parameter

Name	Description
xPos	Pointer to X coordinate of the display origin
yPos	Pointer to Y coordinate of the display origin

Return Value

Name	Description
PPSM_ERR_PAN_INIT	Error when the current task has no panning screen
PPSM_ERROR	If any of *xPos* or *yPos* is 0
PPSM_OK	Successful operation

Hint

The LCD display origin is the co-ordinate of top left corner of the LCD display screen relative to the panning screen origin.

20.6 CursorGetPos

Syntax

STATUS **CursorGetPos**(P_U16 *xPos*, P_U16 *yPos*)

Description

Returns the coordinate (**xPos*, **yPos*) of the hardware cursor of the current task. If this function is called after calling CursorOff(), error will be returned as no more information about cursor exists after calling CursorOff().

Parameter

Name	Description
xPos	Pointer to X coordinate of the top-left corner of the cursor
yPos	Pointer to Y coordinate of the top-left corner of the cursor

Return Value

Name	Description
PPSM_ERR_CURSOR_INIT	Error if cursor is never set or CursorOff() is just called.
PPSM_ERR_PAN_INIT	Error when the cursor is not set
PPSM_ERROR	If any of *xPos* and *yPos* is 0
PPSM_OK	Successful operation

20.7 CursorGetStatus

Syntax

STATUS **CursorGetStatus**(P_U16 *status*)

Description

Returns the status of the hardware cursor of the current task.

Parameter

Name	Description
status	Pointer to status flag of the cursor. It can be one of the following values: ` PPSM_CURSOR_OFF Cursor is OFF ` PPSM_CURSOR_ON Cursor is ON in full density ` PPSM_CURSOR_REVERSED Cursor is ON in reverse video

Return Value

Name	Description
PPSM_ERR_CURSOR_INIT	Error if cursor is never set or CursorOff() is just called.
PPSM_OK	Successful operation

20.8 CursorInit

Syntax

STATUS **CursorInit**(U16 *cursorWidth*, U16 *cursorHeight*)

Description

Changes hardware cursor size to be of width *cursorWidth* and height *cursorHeight*. The width and height must be less than 31 pixels.

Parameter

Name	Description
cursorWidth	Width of the cursor in number of pixels
cursorHeight	Height of the cursor in number of pixels

Return Value

Name	Description
PPSM_ERR_PAN_INIT	Error when the current task has no panning screen
PPSM_ERR_CURSOR_INIT	No more memory to create the hardware cursor information record
PPSM_ERR_WIDTH	If *cursorWidth* is larger than 31
PPSM_ERR_HEIGHT	If *cursorHeight* is larger than 31
PPSM_OK	Successful operation

Hint

If the cursor is large, programmer may use InvRec() to implement a soft cursor.

20.9 CursorOff

Syntax

STATUS **CursorOff**(void)

Description

Turns off the hardware cursor permanently.

Parameter

Name	Description
None	

Return Value

Name	Description
PPSM_ERR_CURSOR_INIT	Error if cursor is never set or CursorOff() is just called.
PPSM_OK	Successful operation

20.10 CursorSet

Syntax

STATUS **CursorSet**(U16 *xPos*, U16 *yPos*)

Description

Sets the top left corner of the hardware cursor to be at (*xPos*, *yPos*). The current task must have panning screen. The (*xPos*, *yPos*) must be within the panning screen. However, it doesn□ check whether the right boundary exceeds the panning screen coordinate.

Parameter

Name	Description
xPos	X coordinate of the top-left corner of the hardware cursor
yPos	Y coordinate of the top-left corner of the hardware cursor

Return Value

Name	Description
PPSM_ERR_PAN_INIT	Current task has no panning screen
PPSM_ERR_CURSOR_INIT	No more memory to create the hardware cursor information record
PPSM_ERR_LCD_X	Invalid x-coordinate
PPSM_ERR_LCD_Y	Invalid y-coordinate
PPSM_OK	Successful operation

20.11 CursorSetBlink

Syntax

STATUS **CursorSetBlink**(U16 *frequency*)

Description

This will set the hardware cursor in blinking mode with *frequency* indicating the number of blinks per 10 seconds. However, the cursor will be seen only if the cursor is set on by calling CursorSetStatus().

Parameter

Name	Description
frequency	blinking rate of the cursor in number of blinks per 10 seconds

Return Value

Name	Description
PPSM_ERR_PAN_INIT	Current task has no panning screen

Name	Description
PPSM_ERR_CURSOR_INIT	No more memory to create the hardware cursor information record
PPSM_ERROR	Error if the frequency cannot be set

20.12 CursorSetOrigin

Syntax

STATUS **CursorSetOrigin**(U16 *xPos*, U16 *yPos*)

Description

Sets the LCD display screen origin to (*xPos*, *yPos*).

Parameter

Name	Description
xPos	X coordinate of the display origin
yPos	Y coordinate of the display origin

Return Value

Name	Description
PPSM_ERR_PAN_INIT	Current task has no panning screen
PPSM_ERR_LCD_X	Invalid x-coordinate
PPSM_ERR_LCD_Y	Invalid y-coordinate
PPSM_OK	Successful operation

Hint

This must be used with LCDScreenMove().

20.13 CursorSetPos

Syntax

STATUS **CursorSetPos**(U16 *xPos*, U16 *yPos*)

Description

Sets the top left corner position of the hardware cursor to be at (*xPos*, *yPos*)

Parameter

Name	Description
xPos	X coordinate of the top-left corner of the cursor
yPos	Y coordinate of the top-left corner of the cursor

Return Value

Name	Description
PPSM_ERR_PAN_INIT	Current task has no panning screen
PPSM_ERR_CURSOR_INIT	No more memory to create the hardware cursor information record
PPSM_ERR_LCD_X	Invalid x-coordinate
PPSM_ERR_LCD_Y	Invalid y-coordinate
PPSM_OK	Successful operation

20.14 CursorSetStatus

Syntax

STATUS **CursorSetStatus**(U16 *status*)

Description

This will set the hardware cursor mode to ON in full density, ON in reverse video mode, or temporary OFF. The hardware cursor will be set on when this function is called with PPSM_CURSOR_ON or PPSM_CURSOR_REVERSED.

Parameter

Name	Description
status	Specify whether a hardware cursor is needed and if so, in what mode it should be: ` PPSM_CURSOR_OFF - temporary no hardware cursor ` PPSM_CURSOR_ON - full density hardware cursor is needed ` PPSM_CURSOR_REVERSED - reverse video mode hardware cursor is needed.

Return Value

Name	Description
PPSM_ERR_PAN_INIT	Current task has no panning screen
PPSM_ERR_CURSOR_INIT	No more memory to create the hardware cursor information record
PPSM_OK	Successful operation

Hint

CursorInit() is called to set the hardware cursor width and height which ranges from 0 to 31 in pixels. CursorSetPos() is called to set the co-ordinate of the top left corner of the hardware cursor. Then finally, CursorSetStatus is called to turn the hardware cursor on. After CursorSetStatus() is called and hardware cursor is on, CursorSetPos() and CursorInit() may be called to change the hardware cursor position or size with immediate effect without calling CursorSetStatus() again.

20.15 DisplayMove

Syntax

STATUS **DisplayMove(**U16 xPos, U16 yPos)

Description

This function is to replace the calling of LCDScreenMove() and CursorSetOrigin(). Whenever the user wants to display a region of panning screen with top left corner of the display at (xPos, yPos) of the panning screen, DisplayMove(xPos, yPos) should be called.

Parameter

Name	Description
xPos	X coordinate
yPos	Y coordinate

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_PAN_INIT	Error when the current task has no panning screen
PPSM_ERR_COORDINATE	Error when any part of LCD is going to display the region outside panning screen

20.16 DrawArc

Syntax

STATUS **DrawArc**(U16 *greyLevel*, U16 *x1*, U16 *y1*, U16 *x2*, U16 *y2*, U16 *style*)

Description

Draws an arc from (*x1*, *y1*) to (*x2*, *y2*).

The arc is actually a quarter of an ellipse with center at (*x2*, *y1*) on which both (*x1*, *y1*) and (*x2*, *y2*) lie.

If dot width is greater than 1, a thick arc will be drawn.

If both fill pattern mode and border mode are set, those area inside arc which is not covered by the border of the arc will be filled.

If fill pattern is set and border is off, those area inside and on the arc border will be filled.

Parameter

Name	Description
greyLevel	Grey level of the arc
x1	x-coordinate of the first point
y1	y-coordinate of the first point
x2	x-coordinate of the second point
y2	y-coordinate of the second point
style	Output Style (AND_STYLE, OR_STYLE, EXOR_STYLE or REPLACE_STYLE)

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_LCD_GREY	Invalid grey level value
PPSM_ERR_LCD_X	Invalid x-coordinate
PPSM_ERR_LCD_Y	Invalid y-coordinate
PPSM_ERR_LCD_STYLE	Invalid style

20.17 DrawCircle

Syntax

STATUS **DrawCircle**(U16 *greyLevel*, U16 *xCenter*, U16 *yCenter*, U16 *radius*, U16 *style*)

Description

Draws a circle centered at (*xCenter*, *yCenter*) with *radius* and *style* as specified.

If dot width is greater than 1, a thick circle will be drawn.

If both fill pattern mode and border mode are set, those area inside the circle which is not covered by border will be filled.

If fill pattern mode is set and border mode is off, the area inside and on the circle border will be filled.

Parameter

Name	Description
greyLevel	Grey level of the circle
xCenter	X-coordinate of the center of circle
yCenter	Y-coordinate of the center of circle
radius	radius of the circle in number of pixels
style	Output Style (AND_STYLE, OR_STYLE, EXOR_STYLE or REPLACE_STYLE)

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_LCD_GREY	Invalid grey level value
PPSM_ERR_LCD_X	Invalid x-coordinate
PPSM_ERR_LCD_Y	Invalid y-coordinate
PPSM_ERR_LCD_RADIUS	Invalid radius
PPSM_ERR_LCD_STYLE	Invalid style

20.18 DrawDot

Syntax

STATUS **DrawDot**(U16 *greyLevel*, U16 *xPos*, U16 *yPos*, U16 *style*)

Description

Outputs a dot with grey level *greyLevel* onto the screen at position (*xPos*, *yPos*) with indicated style.

If dot width is 2, a square dot of length 2 will be drawn where the top left corner is (*xPos*, *yPos*). If dot width is greater than 2, a disc with the center at (*xPos*, *yPos*) will be drawn.

Parameter

Name	Description
greyLevel	Grey level of the dot
xPos	X-coordinate of the dot
yPos	Y-coordinate of the dot
style	Output Style (AND_STYLE, OR_STYLE, EXOR_STYLE, or REPLACE_STYLE).

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_LCD_GREY	Invalid grey level value
PPSM_ERR_LCD_X	Invalid x-coordinate
PPSM_ERR_LCD_Y	Invalid y-coordinate
PPSM_ERR_LCD_STYLE	Invalid style

20.19 DrawEllipse

Syntax

STATUS **DrawEllipse**(U16 *greyLevel*, U16 *xCenter*, U16 *yCenter*, U16
xLength, U16 *yLength*, U16 *style*)

Description

Draws an ellipse centered at (*xCenter*, *yCenter*) with the *xLength* as the width in the x-axis, and *yLength* as the height in the y-axis.

If dot width is greater than 1, a thick ellipse will be drawn.

If both fill pattern mode and border mode are set, those area inside ellipse which is not covered by the border will be filled.

If fill pattern mode is set and border mode is off, the area inside and on the ellipse border will be filled.

Parameter

Name	Description
greyLevel	Grey level of the ellipse
xCenter	X-coordinate of the center of ellipse
yCenter	Y-coordinate of the center of ellipse
xLength	The length of the ellipse in x-axis in pixels
yLength	The length of the ellipse in y-axis in pixels
style	Output Style (AND_STYLE, OR_STYLE, EXOR_STYLE or REPLACE_STYLE)

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_LCD_GREY	Invalid grey level value
PPSM_ERR_LCD_X	Invalid x-coordinate
PPSM_ERR_LCD_Y	Invalid y-coordinate
PPSM_ERR_LCD_STYLE	Invalid style

20.20 DrawHorz

Syntax

STATUS **DrawHorz**(U16 *greyLevel*, U16 *xSrc*, U16 *ySrc*, U16 *width*, U16 *dotLine*, U16 *style*)

Description

Draws a horizontal line from (*xSrc*, *ySrc*) to the right for *width* dots.

If dot width is greater than 1, a thick horizontal line will be drawn. If dot width is greater than 1 and the width of the horizontal line is 1, a square dot of size indicated by dot width will be drawn.

If *dotLine* is non-zero, *dotLine* number of dots will be drawn with the specified grey level; then, the *dotLine* number of dots will be skipped; then, the *dotLine* number of dots will be drawn; and so on.

Parameter

Name	Description
greyLevel	Grey level of the line

Name	Description
xSrc	X-coordinate of the left end-point of the line
ySrc	Y-coordinate of the left end-point of the line
width	The length of the line in pixels
dotLine	Dotted line drawing. This argument accepts a number which represents an equal number of solid dots and skipped dots during line drawing. A value of 0 represents a solid line.
style	Output Style (AND_STYLE, OR_STYLE, EXOR_STYLE, or REPLACE_STYLE).

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_LCD_GREY	Invalid grey level value
PPSM_ERR_LCD_X	Invalid x-coordinate
PPSM_ERR_LCD_Y	Invalid y-coordinate
PPSM_ERR_LCD_STYLE	Invalid style

20.21 DrawLine

Syntax

STATUS **DrawLine**(U16 *greyLevel*, U16 *xSrc*, U16 *ySrc*, U16 *xDest*, U16 *yDest*, U16 *dotLine*, U16 *style*)

Description

Draws a line from (*xSrc*, *ySrc*) to (*xDest*, *yDest*).

If dot width is greater than 1, a thick line will be drawn. If dot width is greater than 1 and (*xSrc*, *ySrc*) equals (*xDest*, *yDest*), a square dot of size indicated by dot width will be drawn.

If *dotLine* is non-zero, *dotLine* number of dots will be drawn with the specified grey level; then, the *dotLine* number of dots will be skipped; then, the *dotLine* number of dots will be drawn; and so on.

Parameter

Name	Description
greyLevel	Grey level of the line
xSrc	X-coordinate of the source point
ySrc	Y-coordinate of the source point
xDest	X-coordinate of the destination point
yDest	Y-coordinate of the destination point
dotLine	Dotted line drawing. This argument accepts a number which represents an equal number of solid dots and skipped dots during line drawing. A value of 0 represents a solid line.
style	Output Style (AND_STYLE, OR_STYLE, EXOR_STYLE, or REPLACE_STYLE).

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_LCD_GREY	Invalid grey level value
PPSM_ERR_LCD_X	Invalid x-coordinate
PPSM_ERR_LCD_Y	Invalid y-coordinate
PPSM_ERR_LCD_STYLE	Invalid style

20.22 DrawRec

Syntax

STATUS **DrawRec**(U16 *greyLevel*, U16 *xSrc*, U16 *ySrc*, U16 *xDest*, U16 *yDest*, U16 *dotLine*, U16 *style*)

Description

Draws a rectangular outline with the top-left corner at (*xSrc*, *ySrc*) and bottom-right corner at (*xDest*, *yDest*).

If dot width is greater than 1, a thick rectangle will be drawn.

If both fill pattern mode and border mode are set, those area inside the rectangle which is not covered by the border will be filled.

If fill pattern mode is set and border mode is off, the area inside and on the rectangle border will be filled.

Parameter

Name	Description
greyLevel	Grey level of the rectangle
xSrc	X-coordinate of the top-left corner
ySrc	Y-coordinate of the top-left corner
xDest	X-coordinate of the bottom-right corner
yDest	Y-coordinate of the bottom-right corner
dotLine	Dotted line drawing. This argument accepts a number which represents an equal number of solid dots and skipped dots during line drawing. A value of 0 represents a solid line.
style	Output Style (AND_STYLE, OR_STYLE, EXOR_STYLE, or REPLACE_STYLE).

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_LCD_GREY	Invalid grey level value
PPSM_ERR_LCD_X	Invalid x-coordinate
PPSM_ERR_LCD_Y	Invalid y-coordinate
PPSM_ERR_LCD_STYLE	Invalid style

20.23 DrawVector

Syntax

STATUS **DrawVector**(U16 *greyLevel*, U16 *numberOfPoints*, P_POINT *pPoints*, U16 *style*, U16 *mode*)

Description

Draws lines to connect points according to the sequence of the data points input. No connection will be made between the first and last points unless it is specified by *mode* or when the last point is the same as the first point.

DrawVector() does not support pattern fill.

Parameter

Name	Description
greyLevel	Grey level of the lines
numberOfPoints	Number of points in the list
pPoints	Pointer to the list of points to be connected
style	Output style can be: ` EXOR_STYLE ` OR_STYLE ` AND_STYLE ` REPLACE_STYLE
mode	Mode should be set to TRUE if the first and last points need to be connected; otherwise, it should be FALSE

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERROR	*numberOfPoints* is 0
PPSM_ERR_LCD_GREY	Invalid grey level
PPSM_ERR_LCD_STYLE	Invalid style
PPSM_ERR_LCD_X	Invalid x-coordinate
PPSM_ERR_LCD_Y	Invalid y-coordinate

Hint

If the output style is EXOR_STYLE, special care should be taken regarding overlapped points.

20.24 DrawVert

Syntax

STATUS **DrawVert**(U16 *greyLevel*, U16 *xSrc*, U16 *ySrc*, U16 *height*, U16 *dotLine*, U16 *style*)

Description

Draws a vertical line from (*xSrc*, *ySrc*) down for *height* dots.

If dot width is greater than 1, a thick vertical line will be drawn. If dot width is greater than 1 and the height of the vertical line is 1, a square dot of size indicated by dot width will be drawn.

If *dotLine* is non-zero, *dotLine* number of dots will be drawn with the specified grey level; then, the *dotLine* number of dots will be skipped; then, the *dotLine* number of dots will be drawn; and so on.

Parameter

Name	Description
greyLevel	Grey level of the line
xSrc	X-coordinate of the top end-point of the line
ySrc	Y-coordinate of the top end-point of the line
height	The length of the line in pixels
dotLine	Dotted line drawing. This argument accepts a number which represents an equal number of solid dots and skipped dots during line drawing. A value of 0 represents a solid line.
style	Output Style (AND_STYLE, OR_STYLE, EXOR_STYLE, or REPLACE_STYLE).

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_LCD_GREY	Invalid grey level value
PPSM_ERR_LCD_X	Invalid x-coordinate
PPSM_ERR_LCD_Y	Invalid y-coordinate
PPSM_ERR_LCD_STYLE	Invalid style

20.25 ExchangeRec

Syntax

STATUS **ExchangeRec**(U16 *xSrc*, U16 *ySrc*, U16 *width*, U16 *height*, P_U8 *bitmap*)

Description

Swaps the image in memory pointed to by *bitmap* with the image at the specified location of the panning screen. The image pointed to by *bitmap* will now be displayed while the rectangular region on the panning screen will be stored at *bitmap*.

Note that the image stored in *bitmap* must be the same size as the specified rectangle in the arguments.

Parameter

Name	Description
xSrc	Top left x-coordinate of the rectangular image to be stored
ySrc	Top left y-coordinate of the rectangular image to be stored
width	Width of the rectangular image to be stored in pixels
height	Height of the rectangular image to be stored in pixels
bitmap	Bitmap image to be displayed

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_LCD_X	Invalid x-coordinate
PPSM_ERR_LCD_Y	Invalid y-coordinate
PPSM_ERR_WIDTH	Invalid width
PPSM_ERR_HEIGHT	Invalid height

20.26 GetDisplayX

Syntax

U16 **GetDisplayX**(void)

Description

Returns the LCD display screen width in number of pixels.

Parameter

Name	Description
None	

Return Value

Name	Description
N/A	The display screen width in pixels

Hint

Application programmers should make use of this function to make the applications size independent.

20.27 GetDisplayY

Syntax

U16 **GetDisplayY**(void)

Description

Returns the display screen height in number of pixels.

Parameter

Name	Description
None	

Return Value

Name	Description
N/A	The display screen height in pixels

Hint

Application programmers should make use of this function to make the applications size independent.

20.28 GetLogicalX

Syntax

U16 **GetLogicalX**(void)

Description

Returns the current panning screen width in number of pixels. This value gives the size that an application can write to, which may be larger than the LCD display screen (i.e. the whole image might not be displayed on the LCD display screen).

Parameter

Name	Description
None	

Return Value

Name	Description
N/A	The panning screen width in pixels

20.29 GetLogicalY

Syntax

U16 **GetLogicalY**(void)

Description

Returns the current panning screen height in number of pixels. This value gives the size that an application can write to, which may be larger than the LCD display screen (i.e. the whole image might not be displayed on the LCD display screen).

Parameter

Name	Description
None	

Return Value

Name	Description
N/A	The panning screen height in pixels

20.30 GetScreenMem

Syntax

P_VOID **GetScreenMem**(U16 *width*, U16 *height*)

Description

This tool will allocate appropriate memory for a panning screen of size *width* and *height*.

Parameter

Name	Description
width	Panning screen width in pixels
height	Panning screen height in pixels

Return Value

Name	Description
N/A	Returns an address for panning screen use or 0 if there is an error

20.31 InvRec

Syntax

STATUS **InvRec**(U16 *xSrc*, U16 *ySrc*, U16 *width*, U16 *height*)

Description

Inverts the grey level of the specified rectangular area on the panning screen.

Parameter

Name	Description
xSrc	Top left x-coordinate of the rectangular area
ySrc	Top left y-coordinate of the rectangular area
width	Width of the rectangular area in pixels
height	Height of the rectangular area in pixels

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_LCD_X	Invalid LCD x-coordinate
PPSM_ERR_LCD_Y	Invalid LCD y-coordinate
PPSM_ERR_WIDTH	Invalid width
PPSM_ERR_HEIGHT	Invalid height

20.32 LCDContrast

Syntax

STATUS **LCDContrast**(U8 contrast)

Description

This tool will pass the value of *contrast* to the contrast control PWM register. The user needs to set or reset the contrast control PWM enable bit.

Parameter

Name	Description
contrast	An 8 bit values to pass to PWM

Return Value

Name	Description
PPSM_OK	Successful operation

20.33 LCDRefreshRate

Syntax

STATUS **LCDRefreshRate(**U16 refreshRate, P_U16 refreshRateSet)

Description

This tool will set the LCD refresh rate to *refreshRate* number of frames per second. As the frame rate will be limited by Pixel Clock Divider Register, Refresh Rate Adjustment Register, Screen Width Register, Screen Height Register and PLL Control Register, PPSM will only set the refresh rate to the closest possible value. If *refreshRate* is 0, the LCD module will be off. *refreshRateSet* will be the previous refresh rate. If user wants to know whether the calling of LCDRefreshRate() is successful, LCDRefreshRate() can be called once more to get the current actual refresh rate returned by *refreshRateSet*.

Parameter

Name	Description
refreshRate	Number of frames per second.
refreshRateSet	Pointer to old frame rate before changes.

Return Value

Name	Description
PPSM_OK	Successful operation

20.34 LCDScreenMove

Syntax

STATUS **LCDScreenMove**(U16 *x*, U16 *y*)

Description

This function is replaced by DisplayMove().

Maps the LCD display screen origin to (*x*, *y*). The rectangular portion of the panning screen with top left corner (*x*, *y*) will be displayed on the LCD display screen.

It is assumed that (*x*, *y*) is within the valid range of panning screen coordinates.

Parameter

Name	Description
x	New x-coordinate of the origin of the LCD display screen
y	New y-coordinate of the origin of the LCD display screen

Return Value

Name	Description
PPSM_OK	Successful operation

Hint

This must be used with CursorSetOrigin().

20.35 PutChar

Syntax

STATUS **PutChar**(U16 greyLevel, P_U8 character, U16 xPos, U16 yPos, U16 font, U16 width, U16 height, U16 style)

Description

This tool will put a 1 bit/pixel font image on 1 bit/pixel or 2 bit/pixel display depending on whether ppsm1.a or ppsm2.a is linked.

Parameter

Name	Description
greyLevel	Grey Level of the character to put on panning screen
character	Pointer to the character bitmap
xPos	x coordinate of the top left corner where font is going to put
yPos	y coordinate of the top left corner where font is going to put
font	SMALL_ITALIC_FONT and LARGE_ITALIC_FONT font will be handled differently to generate the italic effect from a rectangular font bitmap.
width	Width of the character in number of pixels
height	Height of the character in number of pixels

Name	Description
style	It can be REPLACE_STYLE, OR_STYLE,

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_PAN_INIT	Error if the task has no panning screen
PPSM_ERR_GREY	Invalid grey level
PPSM_ERR_LCD_X	Draw outside the panning screen. Invalid x coordinate
PPSM_ERR_LCD_Y	Draw outside the panning screen. Invalid y coordinate
PPSM_ERR_LCD_FONT	Invalid font
PPSM_ERR_LCD_STYLE	Invalid style
PPSM_ERROR	Invalid character pointer

20.36 PutRec

Syntax

STATUS **PutRec**(P_U8 *bitmap*, U16 *xSrc*, U16 *ySrc*, U16 *width*, U16 *height*,
U16 *style*, U16 *reserved*)

Description

Puts a rectangular bitmap image from memory to the specified location of the panning screen.

Parameter

Name	Description
bitmap	Pointer to the bitmap image to be displayed
xSrc	Top left x-coordinate of where the bitmap image will be mapped
ySrc	Top left y-coordinate of where the bitmap image will be mapped
width	Width of the image in pixels
height	Height of the image in pixels
style	Output Style (AND_STYLE, OR_STYLE, EXOR_STYLE or REPLACE_STYLE)

Name	Description
reserved	Reserved for future use

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_LCD_X	Invalid x-coordinate
PPSM_ERR_LCD_Y	Invalid y-coordinate
PPSM_ERR_WIDTH	Invalid width
PPSM_ERR_HEIGHT	Invalid height
PPSM_ERR_LCD_STYLE	Invalid style

20.37 SaveRec

Syntax

STATUS **SaveRec**(P_U8 *bitmap*, U16 *xSrc*, U16 *ySrc*, U16 *width*, U16 *height*,
U16 *reserved*)

Description

Saves a rectangular bitmap image from the specified location on the panning screen to memory.

Parameter

Name	Description
bitmap	Pointer to address where bitmap image is to be saved
xSrc	Top left x-coordinate of the rectangular image area
ySrc	Top left y-coordinate of the rectangular image area
width	Width of the image in pixels
height	Height of the image in pixels
reserved	Reserved for future use

Return Value

Name	Description
PPSM_OK	Successful operation

Name	Description
PPSM_ERR_LCD_X	Invalid x-coordinate
PPSM_ERR_LCD_Y	Invalid y-coordinate
PPSM_ERR_WIDTH	Invalid width
PPSM_ERR_HEIGHT	Invalid height

20.38 SetDotWidth

Syntax

STATUS **SetDotWidth**(U16 *newWidth*, P_U16 *oldWidth*)

Description

This tool sets the width for a dot in number of pixels so that a thick dot can be drawn by DrawDot(), a thick line can be drawn by DrawLine(), etc.

This dot width is applied to DrawDot(), DrawLine(), DrawRec(), DrawCircle(), DrawEllipse(), DrawArc() and DrawVector().

If oldWidth is non-zero, the current dot width will be saved in oldWidth.

Parameter

Name	Description
newWidth	New dot width in pixels
oldWidth	Previous dot width in pixels

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_DOT_WIDTH	*newWidth* is 0

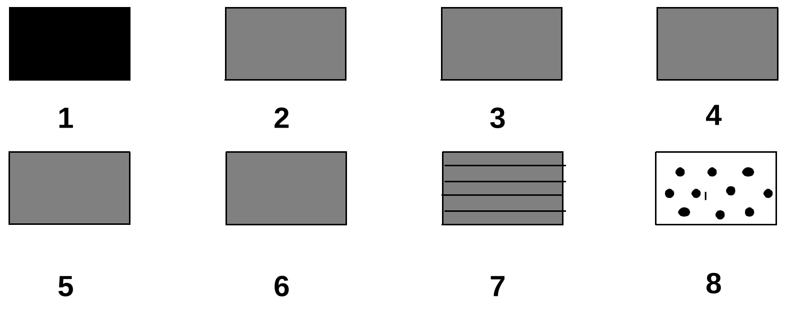
20.39 SetPatternFill

Syntax

STATUS **SetPatternFill**(U16 *mode*, U16 *backGrey*, U16 *borderMode*, U16 *fillSpace*)

Description

This tool sets the pattern fill mode to be applied to DrawRec(), DrawCircle(), DrawEllipse() and DrawArc(). The fill pattern modes are:



The pattern fill mode 0 will turn off the pattern fill feature.

Parameter

Name	Description
mode	There are 8 modes of pattern fill (see description above)
backGrey	The grey level used for the background space
borderMode	Border on-off flag ` TRUE - a border will be drawn around the shape ` FALSE - no border will be drawn around the shape
fillSpace	The gap between the pattern lines. The larger this value is, the more space the pattern will appear.

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_FILL_PATTERN	Invalid fill pattern mode
PPSM_ERR_LCD_GREY	Invalid background grey level
PPSM_ERR_FILL_SPACE	Invalid space gap in fill pattern

Chapter 21 Database Management Tools

21.1 DBAdd

Syntax

STATUS **DBAdd**(P_U32 *dbId*)

Description

Adds a new database for an application to PPSM. This is the first routine to be called whenever a database is created. The database added is a global database.

Parameter

Name	Description
dbId	Returns a database identifier. This identifier is used by the PPSM to refer to this particular database.

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_DB_ADD	Unable to add database

Hint

The returned *dbId* must remain intact. Application uses this as a key to access the database.

21.2 DBAddRecord

Syntax

STATUS **DBAddRecord**(U32 *dbId*, P_U32 *recId*, S32 *numFmt*)

Description

Appends a blank record to the end of the record list for a particular database. User has the option to specify additional formatted data fields to be allocated. The valid range of *numFmt* is from 0 to 5.

Parameter

Name	Description
dbId	Identifier of the database
recId	Returns the identifier of the record added
numFmt	Number of additional formatted data fields in the record

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_DB_DBID	Invalid database identifier
PPSM_ERR_NUM_FMT	Invalid user format field number

Hint

For DBAddRecord() to work, the reference database ID must have been created by calling DBAdd() prior to the calling of DBAddRecord().

21.3 DBAddRecToTop

Syntax

STATUS **DBAddRecToTop**(U32 *dbId*, S32 *numFmt*, P_U32 *outRecId*)

Description

Adds a blank record at the beginning of the record list of a given database. User has the option to specify additional formatted data fields to be allocated. The valid range of *numFmt* is from 0 to 5.

This tool is meant to complement the action of DBAddRecord().

Parameter

Name	Description
dbId	Identifier of the database
numFmt	Number of additional formatted data fields in the record
outRecId	Returns the identifier of the added record

Return value

Name	Description
PPSM_OK	Successful operation
PPSM_ERROR	Unsuccessful operation
PPSM_ERR_DB_DBID	Invalid database identifier
PPSM_ERR_NUM_FMT	Invalid user format field number

Hint

This tool is meant to facilitate the implementation of insertion operation for an ordered record list.

For DBAddRecToTop() to work, the reference database ID must have been created by calling DBAdd() prior to the calling of DBAddRecToTop(). Also, the *recID* passed must be a valid record ID in the database to be operated on.

21.4 DBAppendRecord

Syntax

STATUS **DBAppendRecord**(U32 *dbId*, U32 *recId*, S32 *numFmt*, P_U32 *outRecId*)

Description

Appends a blank record to the record identified by the given *recId* and output the record identifier, *outRecId*, corresponding to the new record. User has the option to specify additional formatted data fields to be allocated. The valid range of *numFmt* is from 0 to 5.

Parameter

Name	Description
dbId	Identifier of the database
recId	Identifier of the record in the record list which the new record is to be appended
numFmt	Number of additional formatted data fields in the new record
outRecId	Returns the record identifier of the new record

Return value

Name	Description
PPSM_OK	Successful operation

Name	Description
PPSM_ERROR	Unsuccessful operation
PPSM_ERR_DB_DBID	Invalid database identifier
PPSM_ERR_DB_RECID	Invalid record identifier
PPSM_ERR_NUM_FMT	Invalid user format field number

Hint

This tool is meant to facilitate the implementation of insertion operation for an ordered record list.

For DBAppendRecord() to work, the reference database ID must have been created by calling DBAdd() prior to the calling of DBAppendRecord(). Also, the *recID* passed must be a valid record ID in the database to be operated on.

21.5 DBChangeStdData

Syntax

STATUS **DBChangeStdData**(U32 *dbId*, U32 *recId*, S32 *fieldId*, P_TEXT *data*)

Description

Changes the data in a predefined standard field of a record in the specified database.

Parameter

Name	Description
dbId	Identifier of the database
recId	Identifier of the record
fieldId	Identifier of the field: ` DB_LAST Last Name ` DB_FIRST First Name ` DB_HOME Home Phone ` DB_OFFICE Office Phone ` DB_ADDRESS Address ` DB_FAX Fax ` DB_COMPANY Company ` 1, 2, 3, 4, 5 Additional fields
data	Data to be put into the field

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERROR	Unsuccessful operation
PPSM_ERR_DB_DBID	Invalid database identifier
PPSM_ERR_DB_RECID	Invalid record identifier
PPSM_ERR_DB_FDID	Invalid field identifier

Hint

Adheres to the size limit of each data field when writing data into it.

The record and database referred to must be valid objects in the PPSM environment.

21.6 DBChangeUnfData

Syntax

STATUS **DBChangeUnfData**(U32 *dbId*, U32 *recId*, S32 *type*, P_U32 *data*,
S32 *size*)

Description

Changes the *data* in the unformatted data portion of a record.

Parameter

Name	Description
dbId	Identifier of the database
recId	Identifier of the record
type	Type of data
data	Starting address of the *data*
size	Size of the *data* (in bytes)

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERROR	Unsuccessful operation
PPSM_ERR_DB_DBID	Invalid database identifier
PPSM_ERR_DB_RECID	Invalid record identifier

Name	Description
PPSM_ERR_DB_TYPE	Invalid data type

Hint

The record and database referred to must be valid objects in the PPSM environment.

Observes the unformatted data type specification for interchangeability.

21.7 DBDelete

Syntax

STATUS **DBDelete**(U32 *dbId*)

Description

Removes a database from PPSM, and frees up all associated memory.

Parameter

Name	Description
dbId	Identifier of the database to be removed

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_DB_DBID	Invalid database identifier

Hint

The database referred to must be a valid object in the PPSM environment.

21.8 DBDeleteRecord

Syntax

STATUS **DBDeleteRecord**(U32 *dbId*, U32 *recId*)

Description

Removes a particular record from the specified database, and frees up all associated memory.

Parameter

Name	Description
dbId	Identifier of the database
recId	Identifier of the record to be removed

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_DB_DBID	Invalid database identifier
PPSM_ERR_DB_RECID	Invalid record identifier

Hint

The database and record referred to must be valid objects in the PPSM environment.

21.9 DBGetFirstRecID

Syntax

STATUS **DBGetFirstRecID**(U32 *dbId*, P_U32 *recId*)

Description

Gets the record ID of the first record in the record list of the given database.

Parameter

Name	Description
dbId	Identifier of the database
recId	Returns the identifier of the top record in the record list

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERROR	Unsuccessful operation
PPSM_ERR_DB_DBID	Invalid database identifier

Hint

This tool is useful for implementing searching on the record list.

The database referred to must be a valid object in the PPSM environment.

21.10 DBGetNextRecID

Syntax

STATUS **DBGetNextRecID**(U32 *dbId*, U32 *recId*, P_U32 *nextID*, P_U16 *botListFlag*)

Description

Gets the identifier of the record following the specified record. If the specified record is the last record, *nextID* returned will be the same as *recId*, and the *botListFlag* will be set.

Parameter

Name	Description
dbId	Identifier of the database
recId	Identifier of specified record
nextID	Returns the identifier of the next record
botListFlag	TRUE if the record identified by *recId* is the last record

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERROR	Unsuccessful operation
PPSM_ERR_DB_DBID	Invalid database identifier
PPSM_ERR_DB_RECID	Invalid record identifier

Hint

This tool is for implementing searching on the record list.

The database and record referred to must be valid objects in the PPSM environment

21.11 DBGetPrevRecID

Syntax

STATUS **DBGetPrevRecID**(U32 *dbId*, U32 *recId*, P_U32 *prevId*, P_U16 *topListFlag*)

Description

Gets the identifier of the record fore-running the specified record. If the specified record is the first record, the *prevId* returned will be the same as *recId*, and the *topListFlag* will be set.

Parameter

Name	Description
dbId	Identifier of the database
recId	Identifier of the specified record
prevID	Returns the identifier of the previous record
topListFlag	TRUE if the record identified by *recId* is the first record

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERROR	Unsuccessful operation
PPSM_ERR_DB_DBID	Invalid database identifier
PPSM_ERR_DB_RECID	Invalid record identifier

Hint

This tool is useful for implementing searching on the record list.

The database and record referred to must be valid objects in the PPSM environment.

21.12 DBReadData

Syntax

STATUS **DBReadData**(U32 *dbId*, U32 *recId*, S32 *fieldId*, P_TEXT **data*)

Description

Reads the formatted *data* from the specified database, record and field.

Parameter

Name	Description
dbId	Identifier of the database
recId	Identifier of the record
fieldId	Identifier of the field
data	Returns the pointer to the starting address of the data

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_DB_DBID	Invalid database identifier
PPSM_ERR_DB_RECID	Invalid record identifier
PPSM_ERR_DB_FDID	Invalid field identifier

Hint

The database and record referred to must be valid objects in the PPSM environment. If a user defined field is to be read, it is the user□ responsibility to check the return status for PPSM_OK, to ensure that the user defined field does exist before using the returned *data* for subsequent processing.

21.13 DBReadTotalNumber

Syntax

STATUS **DBReadTotalNumber** (P_S32 *numDB*)

Description

Reads the total number of databases in the PPSM environment.

Parameter

Name	Description
numDB	Returns the total number of databases

Return Value

Name	Description
PPSM_OK	Successful operation

Name	Description
PPSM_ERR_DB_READNO	Invalid number found

21.14 DBReadTotalNumberRecords

Syntax

STATUS **DBReadTotalNumberRecords**(U32 *dbId*, P_S32 *numRec*)

Description

Reads the total number of records in a given database.

Parameter

Name	Description
dbId	Identifier of the database
numRec	Returns the number of the records in the database

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_DB_DBID	Invalid database identifier

Hint

The database referred to must be a valid object in the PPSM environment.

21.15 DBReadUnfData

Syntax

STATUS **DBReadUnfData**(U32 *dbId*, U32 *recId*, *P_S32 type*, P_U32 **data*,
P_S32 *size*)

Description

Reads the *data* in the unformatted data portion of a record. It will pass back the pointer to the unformatted data being stored in the record. In addition, it will also pass back the *type* and *size* information of the unformatted data.

Parameter

Name	Description
dbId	Identifier of the database
recId	Identifier of the record
type	Returns type of the *data*
data	Returns pointer to the starting address of the data
size	Returns size of the *data* (in bytes)

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERROR	Unsuccessful operation
PPSM_ERR_DB_DBID	Invalid database identifier
PPSM_ERR_DB_RECID	Invalid record identifier

Hint

The database and record referred to must be valid objects in the PPSM environment.

It is good practice for user to check that the return status is PPSM_OK before using the *data*.

21.16 DBRecordSecret

Syntax

STATUS **DBRecordSecret**(U32 *dbId*, U32 *recId*, P_S32 *sFlag*)

Description

Checks if the secret flag of a particular record in a given database is set or not.

Parameter

Name	Description
dbId	Identifier of the database
recId	Identifier of the record

Name	Description
sFlag	Returns the secret flag of the database. It can take either of the following two values: ` 0 Secret flag is cleared ` 1 Secret flag is set

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERROR	Unsuccessful operation
PPSM_ERR_DB_DBID	Invalid database identifier
PPSM_ERR_DB_RECID	Invalid record identifier

Hint

The database and record referred to must be valid objects in the PPSM environment.

21.17 DBSearchData

Syntax

STATUS **DBSearchData**(U32 *dbId*, S32 *fieldId*, P_TEXT *data*, P_U32 *recId*)

Description

Searches though the record list of the given database, finds if there is a record that matches the specified *data* in the specified field. Returns status of operation. If PPSM_OK is returned, the recID passed back is the record identifier of the record with the specified data.

Parameter

Name	Description
dbId	Identifier of the database
fieldId	Identifier of the field
data	Data to be found
recId	Returns the identifier of the record

Return Value

Name	Description
PPSM_OK	Successful operation

Name	Description
PPSM_ERROR	Unsuccessful operation
PPSM_ERR_DB_DBID	Invalid database identifier
PPSM_ERR_DB_RECID	Invalid record identifier
PPSM_ERR_DB_FDID	Invalid field identifier
PPSM_NO_MATCH	No match of data

Hint

Current implementation of tool stop searching once an exact match is found. There is no provision for the case where multiple matches exist.

For good programming practice, user must check the status returned to ensure a valid search is found before using the record identifier returned.

21.18 DBSecretFlag

Syntax

STATUS **DBSecretFlag**(U32 *dbId*, P_S32 *sFlag*)

Description

Checks if the secret flag of a particular database is set or not.

Parameter

Name	Description
dbId	Identifier of the database
sFlag	Returns the secret flag of the database. It can take either of the following two values: ` 0 Secret flag is cleared ` 1 Secret flag is set

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERROR	Unsuccessful operation
PPSM_ERR_DB_DBID	Invalid database identifier

Hint

The database referred to must be a valid object in the PPSM environment.

21.19 DBSetRecordSecretFlag

Syntax

STATUS **DBSetRecordSecretFlag**(U32 *dbId*, U32 *recId*, S32 *sFlag*)

Description

Sets the secret flag of a particular record.

Parameter

Name	Description
dbId	Identifier of the database
recId	Identifier of the record
sFlag	Secret flag of the database. It can take either of the following two values: ` 0 Clears the secret flag ` 1 Sets the secret flag

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERROR	Unsuccessful operation
PPSM_ERR_DB_DBID	Invalid database identifier
PPSM_ERR_DB_RECID	Invalid record identifier
PPSM_ERR_DB_SFLAG	Invalid secret flag value

Hint

The database and record referred to must be a valid object in the PPSM environment.

Note that when a new record is created, its secret flag will be set according to the secret flag of the database which it belongs to. If a user wants all records in the database to be set secret, it should call DBSetSecretFlag() immediately after DBAdd() is called.

21.20 DBSetSecretFlag

Syntax

STATUS **DBSetSecretFlag**(U32 *dbId*, S32 *sFlag*)

Description

Sets the secret flag of a database. If it is set, all new records created subsequently in the specified database will be set to secret.

Parameter

Name	Description
dbId	Identifier of the database
sFlag	Secret flag of the database. It can take either of the following two values: ` 0 Clears the secret flag ` 1 Sets the secret flag

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERROR	Unsuccessful operation
PPSM_ERR_DB_DBID	Invalid database identifier
PPSM_ERR_DB_SFLAG	Invalid secret flag value

Hint

The database referred to must be a valid object in the PPSM environment. User can use this flag to implement data security mechanism at a higher level.

Note that the secret flag is set to 0 when the database is created. If a user wants all records in the database to be set secret, it should call DBSetSecretFlag() immediately after DBAdd() is called.

Chapter 22 Text Tools

22.1 TextCreate

Syntax

STATUS **TextCreate**(P_U32 *templateId*)

Description

Creates and initializes a text template for storing text properties. An identifier for the created text template will be returned.

This is the first text tool an application MUST call before any text can be displayed on the panning screen. The *templateId* returned is required for further references to this particular text template.

Parameter

Name	Description
templateId	Identifier of the newly created text template. This is valid only when PPSM_OK is returned.

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_TEXT_CR	Error while creating text template

22.2 TextDelete

Syntax

STATUS **TextDelete**(U32 *templateId*)

Description

Deletes a text template created by TextCreate(). This should be done when the text template is not needed anymore.

Parameter

Name	Description
templateId	Identifier of the text template to be deleted

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_TEXT_ID	Invalid text template identifier

22.3 TextMap

Syntax

STATUS **TextMap**(U32 *templateId*, P_TEXT *buffer*, U16 *numChar*)

Description

Displays the given text onto the panning screen with properties specified in the text template identified by *templateId*.

The text will be displayed starting at the current character cursor position. The font type, output style and grey level of the text are specified by the text template. Text that extends beyond the size of the text display area specified by the text template will be truncated. The current character cursor position is automatically updated by the system.

Parameter

Name	Description
templateId	Identifier of the text template with current text properties
buffer	Pointer to text string to be displayed
numChar	Number of characters to be displayed

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_TEXT_ID	Invalid text template identifier
PPSM_ERR_TEXT_CUR	Invalid character cursor position

22.4 TextReadCursor

Syntax

STATUS **TextReadCursor**(U32 *templateId*, P_U16 *cursor*)

Description

Reads the current character cursor position of the text template identified by *templateId*. The character cursor position returned is relative to the origin of the text display area.

Parameter

Name	Description
templateId	Text template identifier
cursor	Current character cursor position

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_TEXT_ID	Invalid text template identifier

22.5 TextSetCursor

Syntax

STATUS **TextSetCursor**(U32 *templateId*, U16 *cursor)*

Description

Sets the current character cursor position of the text template identified by *templateId* to the new position as specified. The range of valid character cursor positions to set to is zero through (text display area size in number of characters - 1).

Parameter

Name	Description
templateId	Text template identifier
cursor	The character cursor position to set to

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_TEXT_ID	Invalid text template identifier
PPSM_ERR_TEXT_CUR	Invalid character cursor position

22.6 TextSetDisplay

Syntax

STATUS **TextSetDisplay**(U32 *templateId*, U16 *xPos,* U16 *yPos,* U16 *width*,
U16 *height*, U16 *cursor*)

Description

Sets up the text display layout of the given text template with the corresponding given values. The text display layout comprises of the location and size of a text template. Subsequent text mapped using this text template will be displayed with the new layout.

The text display layout specified must reside within the boundary of the panning screen. The size of the text display area in number of pixels varies with the size of the font type specified in the text template. The range of valid character cursor positions to set to is zero through (text display area size in number of characters - 1).

Parameter

Name	Description
templateId	Identifier of text template to be modified
xPos	X-coordinate of top left corner of text display area
yPos	Y-coordinate of top left corner of text display area
width	Width of text display area in number of columns of characters
height	Height of text display area in number of rows of characters
cursor	Character cursor position within the text display area where text will be displayed next

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_TEXT_ID	Invalid text template identifier
PPSM_ERR_TEXT_X	Text template x-coordinate out of range
PPSM_ERR_TEXT_Y	Text template y-coordinate out of range
PPSM_ERR_TEXT_WIDTH	Given width extends text display area beyond the panning screen

Name	Description
PPSM_ERR_TEXT_HEIGHT	Given height extends text display area beyond the panning screen
PPSM_ERR_TEXT_CUR	Invalid character cursor position

22.7 TextSetFont

Syntax

STATUS **TextSetFont**(U32 *templateId*, P_FONTATTR *pFontAttr*)

Description

Sets up the font attributes of the given text template. Subsequent text mapped using this text template will be displayed with these new font attributes.

Eight font types are currently supported. Small Normal and Small Italic are 8 x 8 pixels English fonts. Large Normal and Large Italic are 16 x 16 pixels English fonts. GB Normal is 16 x 16 Chinese font in GB code format. Chinese Normal is the same as GB Normal (for backward compatibility). BIG5 Normal is 16 x 16 Chinese font in BIG5 code format. BIG5 Variable is a variable size font in BIG5 code format.

Note: Asian fonts are supplied by third parties.

The specified font width and height will only take effect if the BIG5 Variable font type is specified. The minimum and maximum size will depend on the specific font libraries being provided.

The attribute field is for future extensions and should be set to zero.

Parameter

Name	Description
templateId	Identifier of text template to be modified

Name	Description
pFontAttr	Pointer to a text font attributes data structure. Supported font types are: ` SMALL_NORMAL_FONT Small Normal (English) ` SMALL_ITALIC_FONT Small Italic (English) ` LARGE_NORMAL_FONT Large Normal (English) ` LARGE_ITALIC_FONT Large Italic (English) ` GB_NORMAL_FONT GB Normal ` CHINESE_NORMAL_FONT same as GB Normal ` BIG5_NORMAL_FONT BIG5 Normal ` BIG5_VARIABLE_FONT BIG5 Variable

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_TEXT_ID	Invalid text template identifier
PPSM_ERR_TEXT_FONT	Invalid font type
PPSM_ERR_NO_MEMORY	Not enough memory

22.8 TextSetOutlook

Syntax

STATUS **TextSetOutlook**(U32 *templateId*, U16 *outputStyle*, U16 *greyLevel*)

Description

Sets up the output style and grey level of the given text template with the given values. Subsequent text mapped using this text template will be displayed with these new settings.

The output style is defined as the arithmetic operation between the text character bitmap and the image on the panning screen where the character bitmap will be displayed. Five output styles are supported. The text bitmap can replace, OR with, AND with, exclusive OR with, or be inverted and replace the existing image.

Up to four grey levels are currently supported. For a 1 bit per pixel system, the grey levels supported are white and black. For a 2 bits per pixel system, the grey levels supported are white, light grey, dark grey and black.

Parameter

Name	Description
templateId	Identifier of text template to be modified
outputStyle	Output style of text to be displayed: ` REPLACE_STYLE Replace existing image ` OR_STYLE OR with existing image ` AND_STYLE AND with existing image ` EXOR_STYLE Exclusive OR with existing image ` INVERSE_STYLE Invert and Replace existing image
greyLevel	Grey level value of the characters: ` WHITE White color text ` LIGHT_GREY Light grey color text ` DARK_GREY Dark grey color text ` BLACK Black color text

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_TEXT_ID	Invalid text template identifier
PPSM_ERR_TEXT_STYLE	Invalid output style
PPSM_ERR_TEXT_GREY	Invalid grey level value

22.9 TextSetup

Syntax

STATUS **TextSetup**(U32 *templateId*, U8 *fontType*, U8 *outputStyle*, U8 *greyLevel,* U16 *xPos,* U16 *yPos,* U16 *width*, U16 *height*)

Description

Sets up the font type, output style, grey level, location and size of the given text template with the given values. Subsequent text mapped using this text template will be displayed with these new settings.

This tool does not support the variable size of BIG5_VARIABLE_FONT font type and is provided for backward compatibility. If BIG5_VARIABLE_FONT is used, default font size of 16 x 16 is used.

Please refer to TextSetDisplay() (*Section 22.6 - TextSetDisplay*), TextSetFont() (*Section 22.7 - TextSetFont*), and TextSetOutlook() (*Section 22.8 - TextSetOutlook*) for detailed descriptions of the corresponding parameters.

Parameter

Name	Description
templateId	Identifier of text template to be modified
fontType	Font type of text to be displayed: ` SMALL_NORMAL_FONT ` SMALL_ITALIC_FONT ` LARGE_NORMAL_FONT ` LARGE_ITALIC_FONT ` CHINESE_NORMAL_FONT ` BIG5_NORMAL_FONT ` BIG5_VARIABLE_FONT
outputStyle	Output style of text to be displayed: ` REPLACE_STYLE ` OR_STYLE ` AND_STYLE ` EXOR_STYLE ` INVERSE_STYLE
greyLevel	Grey level value of the characters: ` WHITE ` LIGHT_GREY ` DARK_GREY ` BLACK
xPos	x-coordinate of top left corner of text display area
yPos	y-coordinate of top left corner of text display area
width	Width of text display area in number of columns of characters
height	Height of text display area in number of rows of characters

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_TEXT_ID	Invalid text template identifier

Name	Description
PPSM_ERR_TEXT_FONT	Invalid font type
PPSM_ERR_TEXT_STYLE	Invalid output style value
PPSM_ERR_TEXT_GREY	Invalid text grey level value
PPSM_ERR_TEXT_X	Text template x-coordinate out of range
PPSM_ERR_TEXT_Y	Text template y-coordinate out of range
PPSM_ERR_TEXT_WIDTH	Given width extends text display area beyond the panning screen
PPSM_ERR_TEXT_HEIGHT	Given height extends text display area beyond the panning screen

22.10 TextUnmap

Syntax

STATUS **TextUnmap**(U32 *templateId*)

Description

Clears the entire text display area specified by the given text template.

Parameter

Name	Description
templateId	Identifier of text template

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_TEXT_ID	Invalid text template identifier

Chapter 23 Timer Tools

23.1 AlarmClear

Syntax

void **AlarmClear**(void)

Description

Clear all alarms set for current task. This will not affect those alarms set for other tasks.

Parameter

Name	Description
None	

Return Value

Name	Description
None	

23.2 AlarmClearId

Syntax

void **AlarmClearId**(U32 *alarmId*)

Description

This function will clear the alarm in the alarm list specified by *alarmId*. The alarm set can be of any task.

Parameter

Name	Description
alarmId	The identifier of the alarm to be cleared

Return Value

Name	Description
None	

23.3 AlarmRead

Syntax

STATUS **AlarmRead**(P_U16 *year*, P_U16 *month*, P_U16 *day*, P_U16 *hour*, P_U16 *minute*, P_U16 *second*)

Description

Reads the up coming alarm set for the current task in alarm list.

Parameter

Name	Description
year	Pointer to the year value of the alarm
month	Pointer to the month value of the alarm
day	Pointer to the day value of the alarm
hour	Pointer to the hour value of the alarm
minute	Pointer to the minute value of the alarm
second	Pointer to the second value of the alarm

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_NO_ALARM	No alarm is set

23.4 AlarmReadId

Syntax

STATUS **AlarmReadId**(U32 *alarmId,* P_U16 *year,* P_U16 *month*, P_U16 *day*, P_U16 *hour*, P_U16 *minute*, P_U16 *second*)

Description

This function will read the alarm time set in *alarmId*.

Parameter

Name	Description
alarmId	The identifier of the alarm to be read
year	Pointer to the year of the alarm
month	Pointer to the month of the alarm

Name	Description
day	Pointer to the day of the alarm
hour	Pointer to the hour of the alarm
minute	Pointer to the minute of the alarm
second	Pointer to the second of the alarm

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERROR	Error if any of the year, month, day, hour, minute or second is NULL
PPSM_ERR_NO_ALARM	No alarm with the specified *alarmId* found in alarm list

23.5 AlarmSet

Syntax

STATUS **AlarmSet**(U16 *year*, U16 *month,* U16 *day*, U16 *hour*, U16 *minute,* U16 *second*)

Description

Sets the *year*, *month*, *day*, *hour*, *minute* and *second* values of an alarm. Once set, the application will receive a software interrupt message from the system when the specified time is reached.

Parameter

Name	Description
year	The new year for the alarm
month	The new month for the alarm
day	The new day for the alarm
hour	The new hour for the alarm, in 24-hour clock unit
minute	The new minute for the alarm, in 60-minute clock unit
second	The new second for the alarm, in 60-second clock unit

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_YEAR	Invalid year value
PPSM_ERR_MONTH	Invalid month value
PPSM_ERR_DAY	Invalid day value
PPSM_ERR_HOUR	Invalid hour value
PPSM_ERR_MINUTE	Invalid minute value
PPSM_ERR_SECOND	Invalid second value

23.6 AlarmSetId

Syntax

STATUS **AlarmSetId**(P_U32 *alarmId*, U16 *year*, U16 *month*, U16 *date*, U16 *hour*, U16 *minute*, U16 *second*)

Description

Set alarm at specific time and return the alarm id. When the time reaches the alarm time, a message with the alarm id, will be passed to the task. Even if the task is swapped out or the system goes to sleep, the alarm task will still be swapped in and the system will wake up. However, if more than one alarm tasks happen, the earlier the alarm is set, the earlier the task will be swapped in. If alarmId is 0, no alarm id will be given but the alarm will still be set. So if several alarms are set at the same time, the task will first swap to the alarm task which set the alarm first, then the second, etc. However, this version the alarm will stop in the task which set the alarm first only, although the alarm messages are sent to other tasks also.

Parameter

Name	Description
alarmId	The pointer to the alarm id set with specific time
year	Alarm year which must be greater than or equal to 1900
month	Alarm month from 1 to 12
date	Alarm date from 1 to 28, 30 or 31 depending on the month and year values.
hour	Alarm hour from 0 to 23

Name	Description
minute	Alarm minute from 0 to 59
second	Alarm second from 0 to 59

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_YEAR	Invalid year
PPSM_ERR_MONTH	Invalid month
PPSM_ERR_DAY	Invalid date
PPSM_ERR_HOUR	Invalid hour
PPSM_ERR_MINUTE	Invalid minute
PPSM_ERR_SECOND	Invalid second

23.7 DateTimeRead

Syntax

STATUS **DateTimeRead**(P_U16 *year*, P_U16 *month*, P_U16 *day*, P_U16 *hour*,
P_U16 *minute*, P_U16 *second*)

Description

Gets the system date and time.

Parameter

Name	Description
year	Pointer to the year value
month	Pointer to the month value
day	Pointer to the day value
hour	Pointer to the hour value
minute	Pointer to the minute value
second	Pointer to the second value

Return Value

Name	Description
PPSM_OK	Successful operation

Name	Description
PPSM_ERR_YEAR	Invalid year pointer
PPSM_ERR_MONTH	Invalid month pointer
PPSM_ERR_DAY	Invalid day pointer
PPSM_ERR_HOUR	Invalid hour pointer
PPSM_ERR_MINUTE	Invalid minute pointer
PPSM_ERR_SECOND	Invalid second pointer

23.8 DateTimeSet

Syntax

STATUS **DateTimeSet**(U16 *year*, U16 *month*, U16 *day*, U16 *hour*, U16 *minute*, U16 *second*)

Description

Sets the system date and time.

Parameter

Name	Description
year	Input year value, starts from 1900
month	Input month value, in range 1 - 12
day	Input day value, in range 1 - 31
hour	Input hour value, in range 0 - 23
minute	Input minute value, in range 0 - 59
second	Input second value, in range 0 - 59

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_YEAR	Invalid year value
PPSM_ERR_MONTH	Invalid month value
PPSM_ERR_DAY	Invalid day value
PPSM_ERR_HOUR	Invalid hour value
PPSM_ERR_MINUTE	Invalid minute value
PPSM_ERR_SECOND	Invalid second value

23.9 DeleteTimer

Syntax

STATUS **DeleteTimer**(U32 *timerId*)

Description

Delete the timer in timer list specified by *timerId*. Timer can be deleted in any task as far as the timer identifier is known.

Parameter

Name	Description
timerId	The timer identifier returned after calling TimeoutId(), RefFineTimeAlarmId() or RefTimeAlarmId().

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERROR	If the *timerId* is not valid.

23.10 InputTimeout

Syntax

STATUS **InputTimeout**(U32 *millisecond*)

Description

Sets the repetitive time-out period for data input from the touch panel. This time-out routine is an explicit time-out routine dedicated for the pen input device.

Once this time-out routine is activated, the time-out period specified in the argument list is set and count down begins immediately after pen up condition is detected. If the time-out period expires before the next pen down has occurred, PPSM will generate a timer interrupt to the calling application; otherwise, the time-out period is reset ready for the next pen input sequence. This is a repetitive time-out that will continuously set the timer for time-out after each pen input stroke, until it is disabled.

To disable the time-out, call this function with zero as the argument. Maximum value allowed is 1000.

In IrptGetData(), the alarm id read will be 0xFFFFFFFF to distinguish this input timeout from normal timeout.

For DragonBall not DragonBall-EZ, the InputTimeout() is also limited by the pen input sampling rate. A higher sampling rate is more likely to give a more accurate result.

Parameter

Name	Description
millisecond	Time-out period in units of millisecond. If this value is zero, time-out is disabled.

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_TMOUT_VALUE	Invalid time-out period

23.11 RefFineTimeAlarm

Syntax

STATUS **RefFineTimeAlarm**(U32 *alarmTime*)

Description

Sets up an alarm time with respect to the current reference timer in unit of 100 microseconds. Maximum period between alarm time and current time is 0x7FFFFFFF/10 millisecond.

Parameter

Name	Description
alarmTime	The absolute value of the alarm time with respect to the current reference timer value in unit of 100 microseconds. Maximum alarm time period is 0x7FFFFFFF/10 milliseconds from the current reference time.

Return Value

Name	Description
PPSM_OK	Successful operation

23.12 RefFineTimeAlarmId

Syntax

STATUS **RefFineTimeAlarmId**(P_U32 *alarmId*, U32 *alarmTime*)

Description

Sets up an alarm time with respect to the current reference timer in unit of 100 microseconds. Maximum period between alarm time and current time is 0x7FFFFFFF/10 milliseconds. The *timerId* will be returned in IrptGetData().

Parameter

Name	Description
alarmId	Pointer to reference timer alarm identifier
alarmTime	The absolute value of the alarm time with respect to the current reference timer value in unit of 100 microseconds.

Return Value

Name	Description
PPSM_OK	Successful operation

23.13 RefFineTimeDiff

Syntax

U32 **RefFineTimeDiff**(U32 *beginTime,* U32 *endTime*)

Description

This routine takes in two reference times and return to the caller the difference between the two times. This routine takes care of wrapped around condition of the 32-bit continuous reference value. All reference timer values in this routine are in unit of 100 microseconds.

Parameter

Name	Description
beginTime	The begin reference time in unit of 100 microseconds
endTime	The end reference time in unit of 100 microseconds

Return Value

Name	Description
N/A	The elapsed time between the two given reference times in unit of 100 microseconds

23.14 RefFineTimeRead

Syntax

U32 **RefFineTimeRead**(void)

Description

Returns the current 32-bit reference timer value to the caller in unit of 100 microseconds. This is recommended to be used to read the reference timer as it doesn□involve much calculation and simply return the value. RefTimeRead() will need more CPU time to convert the reference timer to resolution of millisecond.

Parameter

Name	Description
void	

Return Value

Name	Description
N/A	Returns a 32-bit reference timer value in unit of 100 microseconds

23.15 RefTimeAlarm

Syntax

STATUS **RefTimeAlarm**(U32 *alarmTime*)

Description

Sets up an alarm time with respect to the current reference timer in unit of milliseconds. Maximum period between alarm time and current time is 0x7FFFFFFF/10 milliseconds.

Parameter

Name	Description
alarmTime	The absolute value of the alarm time with respect to the current reference timer value in unit of milliseconds. Maximum alarm time period is 0x7FFFFFFF/10 seconds from the current reference time.

Return Value

Name	Description
PPSM_OK	Successful operation

23.16 RefTimeAlarmId

Syntax

STATUS **RefTimeAlarmId**(P_U32 *alarmId*, U32 *alarmTime*)

Description

Sets up an alarm time with respect to the current reference timer in unit of milliseconds. Maximum period between alarm time and current time is 0x7FFFFFFF/10 milliseconds. The *alarmId* will be returned in IrptGetData().

Parameter

Name	Description
alarmId	Pointer to reference timer alarm identifier
alarmTime	The absolute value of the alarm time with respect to the current reference timer value in unit of milliseconds.

Return Value

Name	Description
PPSM_OK	Successful operation

23.17 RefTimeDiff

Syntax

U32 **RefTimeDiff**(U32 *beginTime,* U32 *endTime*)

Description

This routine takes in two reference times and return to the caller the difference between the two. This routine takes care of wrapped around condition of the 32-bit continuous reference value. All reference timer values in this routine are in unit of milliseconds.

Parameter

Name	Description
beginTime	The begin reference time in unit of millisecond
endTime	The end reference time in unit of millisecond

Return Value

Name	Description
N/A	The elapsed time between the two given reference times in unit of milliseconds

23.18 RefTimeRead

Syntax

U32 **RefTimeRead**(void)

Description

Returns the current 32-bit reference timer value to the caller in unit of milliseconds. The calling of this function will occupy longer CPU time than RefFineTimeRead().

Parameter

Name	Description
void	-

Return Value

Name	Description
N/A	Returns a 32-bit reference timer value in unit of milliseconds

23.19 SetPeriod

Syntax

STATUS **SetPeriod**(U16 *period*)

Description

Sets the periodic interrupt from the system. This routine will cause PPSM to send a periodic interrupt message with the specified duration between messages, to the calling application.

Parameter

Name	Description
period	The period of interrupt can have any one of the following values: ` RTC_PERI_NONE Disable periodic interrupt ` RTC_PERI_SECOND Interrupt per second ` RTC_PERI_MINUTE Interrupt per minute

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_PERIOD	Invalid time-out period

23.20 SetPeriodId

Syntax

STATUS **SetPeriodId**(P_U32 *alarmId,* U16 *period*)

Description

This function will set the periodic interrupt for current task. Hour periodic interrupt is available for MC68EZ328 but not MC68328. If the periodic alarm is going to be killed with RTC_PERI_NONE, RTC_PERI_NO_SECOND, etc., The returned value of the *alarmId* is going to be meaningless.

Parameter

Name	Description
alarmId	The pointer the alarm id set with specific period

Name	Description
period	It can be: RTC_PERI_NONE - Kill all period interrupts in all tasks RTC_PERI_SECOND - Set second periodic interrupts for current task RTC_PERI_MINUTE - Set minute periodic interrupts for current task RTC_PERI_HOUR - Set hour periodic interrupts for current task which is only available for EZ328 RTC_PERI_MIDNIGHT - Set midnight periodic interrupts for current task RTC_PERI_NO_SECOND - Kill second periodic interrupt for current task RTC_PERI_NO_MINUTE - Kill minute interrupt for current task RTC_PERI_NO_HOUR - Kill hour interrupt for current task which is only available for EZ328 RTC_PERI_NO_MIDNIGHT - Kill midnight interrupt for current task

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_PERIOD	Invalid period flag
PPSM_ERR_NO_MEMORY	Out of memory

23.21 Timeout

Syntax

STATUS **Timeout**(U32 *millisecond*)

Description

General timer time-out routine. This is a one-shot time-out.

The application that calls this routine will register a time-out interval with PPSM. A time interrupt is generated to the task when this time-out period is expired.

Maximum allowed value for Timeout() is 1000.

Parameter

Name	Description
millisecond	Time-out period in unit of millisecond. If this value is zero, time-out is disabled.

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_TMOUT_VALUE	Invalid time-out period

23.22 TimeoutId

Syntax

STATUS **TimeoutId**(P_U32 *timerId*, U32 *millisecond*)

Description

General timer time-out routine. This is a one-shot time-out.

The application that calls this routine will register a time-out interval with PPSM. A time interrupt is generated to the task when this time-out period is expired with the timer id. passed to task.

Maximum allowed value for Timeout() is 1000.

Parameter

Name	Description
timerId	Pointer to the timer identifier
millisecond	Time-out period in unit of millisecond. If this value is zero, time-out is disabled.

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_TMOUT_VALUE	Invalid time-out period

Chapter 24 Memory Management Tools

24.1 Lcalloc

Syntax

void ***Lcalloc**(U32 *size*)

Description

Dynamic allocation of run-time memory to the caller. This routine returns to the caller a pointer to a region of free memory within the malloc size specified in the Linker Specification File (.SPC). PPSM returns at least the amount of memory, in number of bytes, that the caller asked for. If there is not enough memory available in the system, a NULL pointer is returned.

The memory pointed to by the pointer is initialized to zero.

Parameter

Name	Description
size	The size of memory required by the caller in number of bytes

Return Value

Name	Description
N/A	zero - Not enough memory in the system non-zero - Pointer to an initialized free memory region

24.2 Lfree

Syntax

void **Lfree**(void **ptr*)

Description

Returns the memory back into the system heap for reuse. The parameter supplied must be a valid pointer obtained from one of the PPSM memory allocation tools.

Parameter

Name	Description
ptr	Pointer to a valid memory location. It must be a pointer returned from one of the PPSM memory allocation tools.

Return Value

Name	Description
None	

24.3 Lmalloc

Syntax

void ***Lmalloc**(U32 *size*)

Description

Dynamic allocation of run-time memory to the caller. This routine returns to the caller a pointer to a region of free memory within the malloc size specified in the Linker Specification File (.SPC). PPSM returns at least the amount of memory, in number of bytes, that the caller asked for. If there is not enough memory available in the system, a NULL pointer is returned.

The memory pointed to by the pointer is un-initialized.

PPSM returns the size of the largest continuous memory block can be used when calling Lmalloc(LARGEST_MALLOC_SIZE).

Parameter

Name	Description
size	The size of memory required by the caller in number of bytes or flag LARGEST_MALLOC_SIZE when inquiring the size of the largest continuous memory block

Return Value

Name	Description
N/A	zero - Not enough memory in the system non-zero - Pointer to an initialized free memory region or the size of the largest continuous memory block

24.4 Lrealloc

Syntax

void ***Lrealloc**(void **ptr*, U32 *size*)

Description

Moving of memory. This routine re-allocates the memory that is being used from one location to another. It allocates a new area, then copies the content at the original location to the new area, and frees the original memory back into the heap. The purpose of this routine is to defragmentize the system memory.

Parameter

Name	Description
ptr	Pointer to a valid memory location. It must be a pointer returned from one of the PPSM memory allocation tools.
size	The size of memory to be reallocated

Return Value

Name	Description
N/A	zero - Not enough memory in the system non-zero - Pointer to a valid free memory region with the original region□ content

24.5 MoveBlock

Syntax

STATUS **MoveBlock**(P_U32 *srcAddr*, P_U32 *destAddr*, U32 *size*)

Description

Copies a block of memory from the specified source location to the specified destination location. The tool automatically detects the direction of movement. Overlapping of memory regions is allowed.

Parameter

Name	Description
srcAddr	Address of source location
destAddr	Address of destination location
size	Size of transfer in bytes

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERROR	Invalid input arguments

24.6 TaskMemUsed

Syntax

STATUS **TaskMemUsed**(U32 *taskId,* P_U32 *pSizeUsed*)

Description

Inquire memory usage of a task. This routine returns to the caller total number of bytes of memory allocated to the given task through Lmalloc(), Lcalloc or Lrealloc().

Parameter

Name	Description
taskId	Identifier of the task
pSizeUsed	Returns the total number of bytes of memory used by the task with the given taskId

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_TASK_ID	Invalid task identifier

24.7 TaskStackAvail

Syntax

S32 **TaskStackAvail**(void)

Description

PPSM returns to the caller the total number of bytes of stack can still be used by current task when calling TaskStackAvail(). Positive returned value indicates stack has not been used up, negative value implies stack has already overflow.

Parameter

Name	Description
None	

Return Value

Name	Description
N/A	Total number of bytes of stack can still be used by current task

24.8 TotalMemSize

Syntax

U32 **TotalMemSize**(void)

Description

This routine returns to the caller the number of bytes of memory can be allocated through Lmalloc(), Lcalloc() or Lrealloc() on the system. Value returned by this function is a constant which is the same as the malloc size specified in the Linker Specification File (.SPC).

Parameter

Name	Description
None	

Return Value

Name	Description
N/A	The number of bytes of mallocable memory on the system

24.9 TotalMemUsed

Syntax

U32 **TotalMemUsed**(void)

Description

Inquire run-time memory usage of the system. This routine returns to the caller the total number of bytes of memory have been allocated to the whole system through Lmalloc(), Lcalloc() or Lrealloc().

Parameter

Name	Description
None	

Return Value

Name	Description
N/A	The total number of bytes of memory allocated to the whole system

Note: Value returned by TotalMemUsed() does not equal to the actual size of all memory resources used by the system. Global variables and strings memory usage are not counted by this function. For SDS user, the initial memory usage can be found by the symbol lister, SYM.EXE, provided in SDS command directory.

Chapter 25 Power Management Tools

25.1 SetDozeMode

Syntax

void **SetDozeMode**(void)

Description

Sets system to go into Doze mode immediately.

Parameter

Name	Description
None	

Return Value

Name	Description
None	

25.2 SetDozePeriod

Syntax

STATUS **SetDozePeriod**(U16 *milliSecond*)

Description

Sets the countdown period to switch the system from Normal mode to Doze mode. A value of zero forces the system to go to Doze mode whenever there is no message to be handled and no task swapping is needed. If *millisecond* equals PPSM_NO_DOZE, no automatically going to doze will be executed and so no automatically going to sleep mode even SetSleepPeriod() is called.

Parameter

Name	Description
milliSecond	Specifies the Doze mode time-out period in unit of milliseconds. The range is from 0 to 60000 milliseconds. The 0 default value means that PPSM enters Doze mode whenever there is no messages and no task swapping is needed. If it□ PPSM_NO_DOZE, no automatically going to doze after doze timeout nor sleep timeout will happen.

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_DOZE_TIME	Doze time-out period out of range

25.3 SetDutyCycle

Syntax

U16 **SetDutyCycle**(U16 *percentage*)

Description

This tool allows the application to set its own duty cycle level. Applications within a system can have different duty cycle percentages. PPSM automatically changes the PCM correspondingly when the application tasks become active.

Parameter

Name	Description
percentage	Specifies the percentage of duty cycle required from the processor core. Range from a minimum of 3% processor usage to a maximum of 100% usage in unit of 3% steps. Anything less than 3% will be set to 3; anything more than 100% will be set to 100. By default, all applications start at 100% duty cycle

Return Value

Name	Description
U16	Returns the previous duty cycle

25.4 SetSleepMode

Syntax

void **SetSleepMode**(void)

Description

Sets system to go into Sleep mode immediately.

Parameter

Name	Description
None	

Return Value

Name	Description
None	

25.5 SetSleepPeriod

Syntax

STATUS **SetSleepPeriod**(U16 *second*)

Description

Sets the countdown period to switch the System from Doze mode to Sleep mode. A value of zero disables the system from going into Sleep mode.

Parameter

Name	Description
second	Specifies the Sleep mode time-out period in unit of seconds, in the range from 0 to 512 seconds. The default value is 0, meaning no Sleep mode is required.

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_SLEEP_TIME	Sleep time-out period out of range

Chapter 26 UART Communication Tools

26.1 UARTConfigure

Syntax

STATUS **UARTConfigure**(U8 *mode,* U16 *baudRate*, U8 *parity*, U8 *stopBits*, U8 *charLen*)

Description

Configures the UART with the given operating *mode*, *baudRate*, *parity*, number of *stopBits*, and *charLen* flag settings.

The UART hardware and the data transmission time-out are reset during the course of this configuration. Any on-going send or receive request are also aborted.

Both the normal NRZ and IrDA operating modes are supported. The minimum and maximum baud rates supported are 300 bits per second and 115200 bits per second correspondingly.

Parameter

Name	Description
mode	Operating mode flag ` UART_NORMAL_MODE Normal NRZ mode ` UART_IRDA_MODE IrDA mode

Name	Description
baudRate	Baud rate flag ` UART_300_BPS 300 bits per second ` UART_600_BPS 600 bits per second ` UART_1200_BPS 1200 bits per second ` UART_2400_BPS 2400 bits per second ` UART_4800_BPS 4800 bits per second ` UART_9600_BPS 9600 bits per second ` UART_14400_BPS 14400 bits per second ` UART_19200_BPS 19200 bits per second ` UART_28800_BPS 28800 bits per second ` UART_38400_BPS 38400 bits per second ` UART_57600_BPS 57600 bits per second ` UART_115200_BPS 115200 bits per second
parity	Parity flag ` NO_PARITY Disable parity ` ODD_PARITY Enable odd parity ` EVEN_PARITY Enable even parity
stopBits	Stop bits flag ` ONE_STOP_BIT One stop bit after a character ` TWO_STOP_BIT Two stop bits after a character
charLen	Character length flag ` SEVEN_BIT_CHAR 7-bit character mode ` EIGHT_BIT_CHAR 8-bit character mode

Return Value

Name	Description
PPSM_OK	Successful operation

Name	Description
PPSM_ERR_INVALID_ACCESS	Invalid access of UART
PPSM_ERR_MODE	Invalid operating mode flag
PPSM_ERR_BAUD	Invalid baud rate flag
PPSM_ERR_PARITY	Invalid parity flag
PPSM_ERR_STOPBIT	Invalid number of stop bits flag
PPSM_ERR_CHARLEN	Invalid character length flag

26.2 UARTFlowCtrl

Syntax

STATUS **UARTFlowCtrl**(U8 *controlType*)

Description

Enable or disable hardware flow control in UART data communication. PPSM can handle data communication with another device by UART with RTS/CTS hardware flow control. An application can enable hardware control by calling UARTFlowCtrl() with appropriate flag.

Parameter

Name	Description
controlType	controlType ` UART_RCTS_ENABLE Enable RTS/CTS hardware flow control ` UART_RCTS_DISABLE Disable RTS/CTS hardware flow control

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_INVALID_ACCESS	Invalid access of UART
PPSM_ERROR	Invalid input argument

26.3 UARTInquire

Syntax

void **UARTInquire**(P_U8 *mode,* P_U32 *baudRate*, P_U8 *parity*, P_U8 *stopBits*, P_U8 *charLen*)

Description

Returns the current operating *mode*, *baudRate*, *parity*, number of *stopBits*, and *charLen* settings of the UART to the calling application.

The returned values, except for *baudRate*, are flag values as described in UARTConfigure() (*Section 26.1 - UARTConfigure*). The baud rate value returned is in unit of bits per second.

Parameter

Name	Description
mode	Current operating mode flag
baudRate	Current baud rate (in units of bps)
parity	Current parity flag
stopBits	Current number of stop bits flag
charLen	Current character length flag

Return Value

Name	Description
None	

26.4 UARTRcvCtrl

Syntax

STATUS **UARTRcvCtrl**(U8 *controlType*)

Description

Pause or continue receiving data through UART from another device. An application can pause or continue data reception through UART when hardware control is enabled. Error message is returned if RTS/CTS hardware flow control is not enabled.

Parameter

Name	Description
controlType	controlType ` UART_RCTS_PAUSE Pause data reception through UART ` UART_RCTS_CONT Continue data reception through UART

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_INVALID_ACCESS	Invalid access of UART
PPSM_ERR_RCTS_IDLE	RTS/CTS hardware flow control is not enabled

26.5 UARTReadData

Syntax

STATUS **UARTReadData**(P_U8 *pData,* U16 *bufSize,* P_U16 *sizeRead*)

Description

Reads data received from the UART.

An application can initiate a receive request to start receiving data from the UART by calling UARTReceive() (*Section 26.6 - UARTReceive*). When PPSM receives data from the UART, it will post an interrupt message to notify the calling application. The calling application can then read the data by calling UARTReadData().

The calling application needs to pass a buffer, with its size, to PPSM for storing the received data. PPSM will pass back to the calling application the actual number of bytes of data read into the application buffer.

An error condition will be returned if the calling application was not granted permission to use the UART, or no receive request has been initiated when UARTReadData() is called.

If RTS/CTS is enabled, RTS pin is negated when PPSM running UARTRead-Data() and asserted after data reading completed.

Parameter

Name	Description
pData	Pointer to buffer for storing received data
bufSize	Size of data buffer (in number of bytes)
sizeRead	Number of bytes of data read

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_INVALID_ACCESS	Invalid access of UART
PPSM_ERR_NO_REQUEST	Receive request was not initiated

26.6 UARTReceive

Syntax

STATUS **UARTReceive**(U8 *receiveFlag*)

Description

Initiates or aborts a UART receive request.

If *receiveFlag* is UART_RECEIVE_REQUEST, the receive request will be initiated, and PPSM will start waiting for incoming data from the UART.

If *receiveFlag* is UART_RECEIVE_ABORT, the on-going receive request will be aborted.

When PPSM receives data, it will post an interrupt message to the calling application notifying it that data is available for it to read. Application can then read the received data by calling UARTReadData() (*Section 26.5 - UARTReadData*).

Application task swapping is disabled while the receive request is in progress, and re-enabled after the receive request is terminated.

Parameter

Name	Description
receiveFlag	Operation flag ` UART_RECEIVE_REQUEST Initiates a receive request ` UART_RECEIVE_ABORT Aborts a receive request

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_INVALID_ACCESS	Invalid access of UART
PPSM_ERR_BUSY	There is already an on-going receive request (when trying to initiate one)

26.7 UARTSend

Syntax

STATUS **UARTSend**(U8 *sendFlag,* P_U8 *pData,* U16 *dataLen*)

Description

Initiates or aborts a UART send request.

If *sendFlag* is UART_SEND_REQUEST, the send request will be initiated, and PPSM will start sending data to the UART.

If *sendFlag* is UART_SEND_ABORT, the on-going send request will be aborted.

The calling application needs to pass the data byte stream, with its size, to PPSM for sending out.

When PPSM finishes sending the data, it will post an interrupt message to the calling application notifying it that all data has been sent.

Application task swapping is disabled while the send request is in progress, and re-enabled after the send request is terminated.

Parameter

Name	Description
sendFlag	Operation flag ` UART_SEND_REQUEST Initiates a send request ` UART_SEND_ABORT Aborts a send request
pData	Pointer to data byte stream
dataLen	Number of bytes of data to be sent

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_INVALID_ACCESS	Invalid access of UART
PPSM_ERR_BUSY	There is already an on-going send request (when trying to initiate one)

26.8 UARTSendAbort

Syntax

STATUS **UARTSendAbort**(U8 *abortFlag*, P_U8 **pSendData*, P_U32 *sendSize*)

Description

Terminate the current UART transmission. The transmission abort process is the same as calling UARTSend(UART_SEND_ABORT). Moreover, beside aborting the ongoing transmission, PPSM returns a pointer which points to the current position of internal transmission buffer and the number of bytes of data have been sent. Caller can get those information without abort UART transmission by calling UARTSendAbort() with UART_INQUIRE_SBYTE.

Parameter

Name	Description
abortFlag	abortFlag ` UART_SEND_ABORT Aborts a send request ` UART_INQUIRE_SBYTE Returns the number of bytes of data have been sent and the position of transmission pointer
pSendData	Position of internal transmission pointer
sendSize	Number of bytes of data have been sent

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_INVALID_ACCESS	Invalid access of UART

26.9 UARTSendCtrl

Syntax

STATUS **UARTSendCtrl**(U8 *controlType*)

Description

Pause or continue sending data through UART to another device. An application can pause or continue data transmission through UART when hardware control is enabled. Error message is returned if RTS/CTS hardware flow control is not enabled.

Parameter

Name	Description
controlType	controlType ` UART_RCTS_PAUSE Pause data transmission through UART ` UART_RCTS_CONT Continue data transmission through UART

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_INVALID_ACCESS	Invalid access of UART
PPSM_ERR_RCTS_IDLE	RTS/CTS hardware flow control is not enabled

26.10 UARTSetDelay

Syntax

STATUS **UARTSetDelay**(U8 *type*, U16 delay)

Description

In order to communicate with application in PC, such as HyperTerminal and Telix, transmitting data in a burst of pulses periodically would greatly increase the accuracy of transmission. This function allows user to set a delay, in unit of 100us, between each transmission of all data in transmit FIFO (between two hardware interrupts).

The range of delay values supported is 1 to 60,000.

Parameter

Name	Description
type	delay type ` UART_TXHALF_DELAY Set a delay between each TXHALF interrupts ` UART_TXDELAY_CLEAR Clear delay within transmission
delay	Transmission delay value in 100 microseconds ` 100 microseconds to 60,000 microseconds (6 seconds)

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_INVALID_ACCESS	Invalid access of UART
PPSM_ERR_INVALID_TXDELAY	Delay value out of range

26.11 UARTTimeout

Syntax

STATUS **UARTTimeout**(U16 *tmout*)

Description

Sets the maximum time interval allowed between two hardware UART interrupts.

This time-out function prevents application from deadlocking itself when the data stream terminates unexpectedly.

System do not initiate a timeout right after UARTTimeout() is called. PPSM starts timeout with the given timeout interval after calling UARTSend(), UARTReceive(), UARTSendCtrl() and UARTRcvCtrl().

The valid range of time-out values can be set is zero through 60,000 (in units of milliseconds). A time-out value of zero means disabling all the time-out function related to UART.

Parameter

Name	Description
tmout	Transmission time-out value in milliseconds ` 0 - to disable time out function ` 1 to 60,000 milliseconds

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_INVALID_ACCESS	Invalid access of UART
PPSM_ERR_INVALID_TMOUT	Time-out value out of range

Chapter 27 Task Handling Tools

27.1 AdvTaskCreate

Syntax

STATUS **AdvTaskCreate**(P_U32 *taskId,* P_VOID *procAddr,* S16 *xSrc,* S16 *ySrc,* S16 *xDest,* S16 *yDest,* S32 *stackSize,* U16 *newScreen,* U16 *screenWidth,* U16 *screenHeight,* P_U8 *bitmap*)

Description

Creation of a new PPSM task. This tool creates a PPSM application task in the same manner as the tool TaskCreate() but with advanced configuration details. It allows the caller to specify:

` The launch icon position and size. Launch icon can be on screen or off screen. For either width or height is zero, no launch icon is going to be created.

` The stack memory size required by the task.

` The panning screen dimensions. PPSM task can have no associated screen, or a screen with user specified dimension.

The default screen dimensions are taken from the linker specification file, LCDVIRTWIDTH and LCDVIRTHEIGHT.

Parameter

Name	Description
taskId	Returns a task identifier. This identifier is used by PPSM to refer to the task when it uses the system resources.
procAddr	Address of the application task
xSrc	Top-left x-coordinate of task icon
ySrc	Top-left y-coordinate of task icon
xDest	Bottom-right x-coordinate of task icon
yDest	Bottom-right y-coordinate of task icon
stackSize	Required stack size in bytes. If a negative number is used, a default of 512 bytes is used.

Name	Description
newScreen	Screen Flag - ` PPSM_SCREEN_NOSCREEN No panning screen is needed ` PPSM_SCREEN_NEW A panning screen of size *screenWidth* by *screenHeight* is required
screenWidth	If *newScreen* == PPSM_SCREEN_NEW, this argument is the panning screen width in number of pixels. If this value is 0, the default screen width and height are used
screenHeight	If *newScreen* == PPSM_SCREEN_NEW, this argument is the panning screen height in number of pixels. If this value is 0, the default screen height and width are used
bitmap	The bitmap for launch icon within LCD display

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_TASK_ID	Invalid address for storing task identifier
PPSM_ERR_TASK_FLAG	Invalid screen flag
PPSM_ERR_TASK_WIDTH	Invalid screen width
PPSM_ERR_TASK_HEIGHT	Invalid screen height
PPSM_ERR_COORDINATE	Invalid coordinates
PPSM_ERR_NO_MEMORY	Not enough memory

27.2 AppSwap

Syntax

void **AppSwap**(U16 *flag*)

Description

If *flag* is FALSE, no task swapping is allowed by any means such as pen down on application icon nor using SendMessage(), etc. If SendMessage() or AdvSendMessage() are called after this function with FALSE in *flag*, the message will still be sent to the target task but the task swapping action will be ignored. Moreover, if several AppSwap(FALSE) are called, the same number of AppSwap(TRUE) must be called before the task swapping is enabled.

Parameter

Name	Description
flag	TRUE - enable task swapping FALSE - disable task swapping

Return Value

Name	Description
None	

27.3 SubTaskCreate

Syntax

STATUS **SubTaskCreate**(P_U32 *taskId*, P_VOID *procAddr*, U16 *stackSize*,
U16 *numArg*, ...)

Description

Creating a sub-task. Any task can use this tool to create sub-tasks. If the calling task is itself a sub-task, the new sub-task will belong to its parent(i.e. the calling sub-task and the newly created sub-task become siblings). If more than one sub-task has been created under a parent, the new sub-task will be added to the head of the sub task list. There is currently no limit on the number of sub-task a parent task can have. However, the order of the sub-task chain may change in run-time.

This routine accepts variable length input arguments. These arguments are passed into the sub-task by PPSM, meaning that the actual sub-task routine can accept input arguments.

Parameter

Name	Description
taskId	Returns a task identifier. This identifier is used by PPSM to refer to the task when it uses the system resources
procAddr	Address of the sub task routine
stackSize	Stack size required for the sub task. If a zero is used, the default of 2K byte is used
numArg	Number of variable arguments. Each argument takes up 4 bytes in the argument stack.

Name	Description
...	Variable arguments. These are passed to the sub task routine when the sub task begins execution

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_NO_MEMORY	Not enough memory

27.4 TaskCreate

Syntax

STATUS **TaskCreate**(P_U32 *taskId*, P_VOID *procAddr*, S16 *xSrc*, S16 *ySrc*, S16 *xDest*, S16 *yDest*, P_U8 *bitmap*)

Description

PPSM needs to know the existence of each application during initialization stage. The main body of a PPSM system must call this routine once for each application. PPSM will create the necessary data structure and memory space required to run the application.

Parameter

Name	Description
taskId	Returns a task identifier. This identifier is used by PPSM to refer to the task when it uses the system resources.
procAddr	Address location of the application
xSrc	Top left x-coordinate of the task icon
ySrc	Top left y-coordinate of the task icon
xDest	Bottom right x-coordinate of the task icon
yDest	Bottom right y-coordinate of the task icon
bitmap	Pointer to bitmap of the task icon ` 0 - No on-screen icon is needed

Return Value

Name	Description
PPSM_OK	Successful operation

Name	Description
PPSM_ERR_TASK_ID	Invalid task identifier
PPSM_ERR_COORDINATE	Invalid coordinates
PPSM_ERR_NO_MEMORY	Not enough memory

27.5 TaskHook

Syntax

STATUS **TaskHook**(U32 taskId, P_VOID entryCallback, P_VOID exitCallback)

Description

Set the entry and exit routines to the specific task. Whenever the task is going to be swapped in, entryCallback function will be executed. Whenever the task is going to be swapped out, exitCallback will be executed. The entryCallback and exitCallback should be function calls with U32 as input parameter and the function should not involve interrupts, e.g. Entry(U32 oldApp) and Exit(U32 newApp) where oldApp will be the task identifier of the task just swapped out and newApp will be the next task to be swapped in.

Parameter

Name	Description
taskId	Task identifier of the task to be hooked with the entry and exit functions.
entryCallback	The entry function to be executed before swapping in the task. If NULL is input, no entry function will be executed.
exitCallback	The exit function to be executed after swapping out the task. If NULL is input, no exit function will be executed.

Return Value

Name	Description
PPSM_OK	Successful Operation
PPSM_ERR_TASK_ID	Invalid task id.

27.6 TaskReInit

Syntax

STATUS **TaskReInit**(U32 *taskId*, U16 *flag*)

Description

To set the specific task to be in reinit mode so that each time the task is swapped in, it will start from beginning of the task again. This function is generally called once the task is created.

Parameter

Name	Description
taskId	The task identifier of the task to be set.
flag	TRUE or FALSE to indicate whether the task needs to be in reinit mode.

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_TASK_ID	Invalid task id.

27.7 TaskStart

Syntax

STATUS **TaskStart**(U32 *taskId*)

Description

Begin execution of the first application task. This routine will never returns. It launches the first PPSM application, and all other applications are started by activating the application icons. This routine is used at the start to kick off the system.

Parameter

Name	Description
taskId	The task identifier for the first application to be launched.

Return Value

Name	Description
PPSM_ERROR	Invalid *taskId*
PPSM_OK	Task started successfully

27.8 TaskTerminate

Syntax

STATUS **TaskTerminate**(U32 *taskId*)

Description

Termination of a task. The task identifier can be of a main or sub task. All system memory, such as stack memory and screen (if any), associated with the task and its subtasks that are allocated by PPSM will be freed.

A task cannot terminate itself nor its parent task if it is a sub-task. TaskTerminate() will not free the memory that has been explicitly allocated by the task with Lmalloc().

Parameter

Name	Description
taskId	The task identifier of the task to be terminated

Return Value

Name	Description
PPSM_OK	Task successfully terminated
PPSM_ERR_TASK_ID	Invalid task identifier

Chapter 28 Inter-Task Messaging Tools

28.1 AdvMessageDelete

Syntax

STATUS **AdvMessageDelete**(P_TASKDESC task,U16 type, U32 shortData)

Description

This function will delete those messages in a task message queue with specific task, type and shortData matched. If any of the parameter value is 0xFFFFFFFF in task and shortData or 0xFFFF in type, it means "don care" in that field. SO if task is 0xFFFFFFFF, all tasks will be check and the matched messages in all tasks will be deleted, etc.

Parameter

Name	Description
task	Task identifier
type	The message type such as IRPT_UART, etc.
shortData	This field is generally returned in IrptGetData() as alarmId, timerId, areaId, taskId, etc.

Return Value

Name	Description
PPSM_ERR_TASK_ID	Invalid task id.
PPSM_OK	Successful Operation

28.2 AdvSendMessage

Syntax

STATUS **AdvSendMessage**(U32 *taskId*, P_MESSAGE *msg, U8 flag*)

Description

This tool is used when the current task wants to send a message to another task. If the receiver task task identifier is not known, this tool cannot be used.

If msg is 0, no message will be sent but the task swapping may still be executed.

All data that the sender wants to send must be stored in the form of MESSAGE structure. No protocol or data format is put on the message data. The sender and receiver must have a mutual understanding of the data representation of the message being sent.

The data structure for the structure MESSAGE is:

```
typedef struct _MESSAGE
        {
                U16      messageType;            /*  message type  */
                U16      message;                /*  message  */
                U32      misc;                   /*  short data (32bit)  */
                P_VOID data;                     /*  associated data, if any */
                U16      size;                   /*  size of data in bytes  */
                U16      reserved;               /*  for future  (broadcast, etc) */
        } PPSM_MESSAGE, *P_MESSAGE;
```

If AppSwap(FALSE) is called before calling this function, the message will still be sent but any form of task swapping action will be ignored.

If the system is in doze mode, calling this function will wake up the system.

Parameter

Name	Description
taskId	The receiver task identifier
msg	The message to send. All data to send are stored in the PPSM_MESSAGE structure, with the following representation: ` messageType - Must set to MESSAGE_IRPT. ` message - The type of message being sent to the receiver. Normally set to IRPT_USER. ` misc - 32-bit short data ` data - data pointer to the buffer that is storing the message data ` size - size of data buffer, in number of bytes ` reserved - not used
flag	It can be: SWAP_TASK_LATER - Task swapping will happen in IrptGetData() when all messages in current task are handled. SWAP_TASK_BACK_LATER - Task swapping will happen immediately and the current task will be swapped back when all messages in the target task are handled. SWAP_TASK - Task swapping will happen immediately. NO_SWAP_TASK - No task swapping will happen.

Return Value

Name	Description
PPSM_OK	Message successfully sent
PPSM_ERR_TASK_ID	Invalid task identifier
PPSM_ERR_NO_MEMORY	Not enough memory
PPSM_ERROR	Invalid flag or AppSwap(FALSE) is called before calling this function

28.3 MessageDelete

Syntax

STATUS **MessageDelete**(U16 type)

Description

This function will delete those messages in current task□ message queue with specific type matched. If the input parameter value is 0xFFFF in type, it means "don□care" in that field and all messages in current task will be deleted.

Parameter

Name	Description
type	The message type such as IRPT_UART, etc.

Return Value

Name	Description
PPSM_ERR_TASK_ID	Invalid task id.
PPSM_OK	Successful Operation

28.4 SendMessage

Syntax

STATUS **SendMessage**(U32 *taskId*, P_MESSAGE *msg*)

Description

This tool is used when the current task wants to send a message to another task. If the receiver task□ task identifier is not known, this tool cannot be used.

All data that the sender wants to sent must be stored in the form of MESSAGE structure. No protocol or data format is put on the message data. The sender and receiver must have a mutual understanding of the data representation of the message being sent. On the receiving side, IrptGetData() is where the message received.

The data structure for the structure MESSAGE is:

```
typedef struct _MESSAGE
      {
            U16   messageType;       /*  message type  */
            U16   message;           /*  message  */
            U32   misc;              /*  short data (32bit)  */
            P_VOID data;             /*  associated data, if any */
            U16   size;              /*  size of data in bytes  */
            U16   reserved;          /*  for future  (broadcast, etc) */
      } PPSM_MESSAGE, *P_MESSAGE;
```

If AppSwap(FALSE) is called before calling this function, the message will still be sent but any form of task swapping action will be ignored.

If the system is in doze mode, calling this function will wake up the system.

Parameter

Name	Description
taskId	The receiver task□ identifier
msg	The message to send. All data to send are stored in the PPSM_MESSAGE structure, with the following representation: ` messageType - Must set to MESSAGE_IRPT. ` message - The type of message being sent to the receiver. Normally set to IRPT_USER. ` misc - 32-bit short data ` data - data pointer pointing a buffer that is storing the message data ` size - size of data buffer, in number of bytes ` reserved - not used

Corresponding values between SendMessage() and IrptGetData()

SendMessage	IrptGetData
STATUS **SendMessage**(U32 *taskId*, P_MESSAGE *msg*) U16 **IrptGetData**(P_U32 *sData*, P_U32 **data*, P_U32 *size*)	
msg.messageType	must be MESSAGE_IRPT
msg.message	returned value
msg.misc	*sData

SendMessage	IrptGetData
msg.data	data
msg.size	size
msg.reserved	N/A

Return Value

Name	Description
PPSM_OK	Message successfully sent
PPSM_ERR_TASK_ID	Invalid task identifier
PPSM_ERR_NO_MEMORY	Not enough memory
PPSM_ERROR	AppSwap(FALSE) is called before calling this function

Chapter 29 Interrupt Handling Tools

29.1 IrptGetData

Syntax

U16 **IrptGetData**(P_U32 *sData*, P_U32 **data*, P_U32 *size*)

Description

This tool reads the application`s interrupt buffer for any pending interrupt message. The interrupt source of the message and data collected from the interrupt handler is returned. Each application has its own unique interrupt buffer.

The data returned from this routine depends on the nature of the interrupt type. Different messages from different interrupt sources represent different types of data. The pre-defined format for the data generated by system interrupts are listed below. The size of the data collected during an interrupt event is returned in the last argument *size*. It is in number of byte.

PPSM does not impose any format or protocol for message data generated from User Defined handlers since such data is programmable by the system integrator.

Parameter

Return Value	*sData	*(*data)	*size
IRPT_AUDIO	N/A	U16 - AUDIO_STOP_WAVE or AUDIO_STOP_TONE	2
IRPT_HWR	N/A	U16, ... - List of candidates	2 x no. of candidat es
IRPT_ICON	U32 - AreaId	U16 - PPSM_ICON_TOUCH or PPSM_ICON_DRAG or PPSM_ICON_PEN_UP or PPSM_ICON_DRAG_UP	2
IRPT_INPUT_ STATUS	U32 - AreaId	U16 - PPSM_INPUT_TOUCH or PPSM_INPUT_DRAG or PPSM_INPUT_PEN_UP or PPSM_INPUT_DRAG_UP	2
IRPT_KEY	N/A	TEXT, S16, S16 - keycode, (x, y)	6

Return Value	*sData	*(*data)	*size
IRPT_PEN	U32 - AreaId	S16, S16 - (x, y)	4
IRPT_RTC	U32 - TimerId	N/A	0
IRPT_TIMER	U32 - TimerId	N/A	0
IRPT_UART	N/A	U16, U16 - UART_ERROR, UART_ERR_TMOUT or UART_ERR_FRAME or UART_ERR_PARITY or UART_ERR_OVERRUN or UART_ERR_NODATA OR U16 - UART_DATA_RECEIVED or UART_DATA_SENT	4 2
IRPT_USER	User defined	User defined	User defined

Return Value

Name	Description
IRPT_AUDIO	Indicating audio stopped
IRPT_ERROR	Invalid function parameter
IRPT_HWR	Handwriting Recognition interrupt
IRPT_ICON	Pen input on icon active area interrupt
IRPT_INPUT_STATUS	Pen Action Status for pen input
IRPT_INT	INT0-INT7 User Defined Handler
IRPT_IRQ1	IRQ1 User Defined Handler
IRPT_IRQ2	IRQ2 User Defined Handler
IRPT_IRQ3	IRQ3 User Defined Handler
IRPT_IRQ6	IRQ6 User Defined Handler
IRPT_KEY	External and soft keyboard interrupt
IRPT_NONE	No application interrupt has occur
IRPT_PEN	Pen input on application active area interrupt
IRPT_PWM	PWM User Defined Handler
IRPT_RTC	Real Time Clock interrupt

Name	Description
IRPT_SPIM	SPI Master User Defined Handler
IRPT_SPIS	SPI Slave User Defined Handler(DragonBall Only)
IRPT_TIMER	Preset timer interrupt
IRPT_UART	UART User Defined Handler
IRPT_USER	User Defined Handler
IRPT_WDG	Watchdog User Defined Handler

29.2 IrptRelease

Syntax

STATUS **IrptRelease**(U16 *handlerFlag*)

Description

This tool is used by the application to release an interrupt handler that the caller has successfully requested previously. The interrupt handler that is being released must be a valid handler that has been granted to the application via the IrptRequest() tool. If a handler that the application has no hook to is being requested for release, an error message will be returned. When a handler is released, any data or message still pending in the interrupt handler is flushed out and removed.

Once an interrupt handler is released, PPSM can then grant the handler to other applications that request for the handler.

The application should release the handlers one at a time.

Parameter

Name	Description
handlerFlag	Flag to indicate which of the handler the caller is releasing. ` IRPT_SPIM_FLAG ` IRPT_SPIS_FLAG(DragonBall Only) ` IRPT_UART_FLAG ` IRPT_IRQ1_FLAG ` IRPT_IRQ2_FLAG ` IRPT_IRQ3_FLAG ` IRPT_IRQ6_FLAG ` IRPT_INT_FLAG ` IRPT_WDOG_FLAG ` IRPT_PWM_FLAG ` IRPT_USER_FLAG

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_RELEASE	Unable to release handler

29.3 IrptRequest

Syntax

U16 **IrptRequest**(U16 *handlerFlag*)

Description

This tool is used by the application task to request the services of the interrupt handlers. Once requested and granted, all interrupt messages sent from the handlers are directed to the application with the appropriate interrupt identifiers.

When calling this tool, the user must specify which of the handlers it wishes to request. The interrupt flags can be OR`ed together to request more than one handler with a single call. If the requested handlers are installed and available, PPSM system will hook the handlers to the application; if not, PPSM will do nothing.

The return value from this tool is a 16-bit word, returning the flags of the handlers that have been granted, if any. It uses the same format as the input handler flag parameter. For example, if a caller makes a request for a specific set of handlers, and if all are granted, then the return value from the tool will be the same as the input handler flag parameter. If any one of the requested handlers cannot be granted, the return value will be different from the input flag parameter. If none is granted, a zero is returned.

A request is successful if and only if the handler being requested has not been granted to another task.

Parameter

Name	Description
handlerFlag	Flag to indicate which of the handlers the caller is requesting. Each bit of this flag represents a specific handler. The following flag values can be OR□d together if more than one handler is being requested: ` IRPT_SPIM_FLAG ` IRPT_SPIS_FLAG(DragonBall only) ` IRPT_UART_FLAG ` IRPT_IRQ1_FLAG ` IRPT_IRQ2_FLAG ` IRPT_IRQ3_FLAG ` IRPT_IRQ6_FLAG ` IRPT_INT_FLAG ` IRPT_WDOG_FLAG ` IRPT_PWM_FLAG ` IRPT_USER_FLAG

Return Value

Name	Description
N/A	Returns the handlers that has/have been granted. Each bit in the returned 16-bit word represents a specific handler: ` IRPT_SPIM_FLAG ` IRPT_SPIS_FLAG(DragonBall only) ` IRPT_UART_FLAG ` IRPT_IRQ1_FLAG ` IRPT_IRQ2_FLAG ` IRPT_IRQ3_FLAG ` IRPT_IRQ6_FLAG ` IRPT_INT_FLAG ` IRPT_WDOG_FLAG ` IRPT_PWM_FLAG ` IRPT_USER_FLAG

29.4 IrptSendData

Syntax

STATUS **IrptSendData**(U16 *irptType,* U32 *sData,* P_U32 *data,* U32 *size*)

Description

Passes the user defined interrupt message from interrupt handler back to application level. This routine should be called from the user installed interrupt handler only. After this message is sent from the interrupt handler, the application that has requested the handler will be able to receive this message via the IrptGetData() tool.

Parameter

Name	Description
irptType	Interrupt Identifier: ` IRPT_SPIM ` IRPT_SPIS(DragonBall only) ` IRPT_IRQ1 ` IRPT_IRQ2 ` IRPT_IRQ3 ` IRPT_IRQ6 ` IRPT_INT ` IRPT_WDOG ` IRPT_PWM ` IRPT_USER
sData	This field can be used to send 4 bytes or less data to the application
data	Data buffer for storing data to send to the application
size	The size of data being sent, in number of bytes

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_IRPT_HANDLER	Handler not requested by application
PPSM_NO_MEMORY	Not enough memory

Chapter 30 System Tools

30.1 PPSMInit

Syntax

STATUS **PPSMInit**(U16 *calibration*)

Description

PPSM initialization routine. This routine must be called at the beginning of the system file before any of the PPSM tools can be used.

The input argument allows the system caller to decide if pen calibration is required at this time.

With the default calibration device driver, two cross-hairs will be drawn on the screen, one near the top-right corner and the other near the bottom-left corner. The user must press the center of these cross-hairs one at a time to perform the pen input calibration.

The default calculation value is based on a LCD that has 320 pixels by 240 pixels (physical sizes 12 cm by 9 cm), using a 10-bit A/D convertor. An offset of 100 (in A/D output unit) is chosen as the A/D non-linear area around the edge of the touch panel. User may define his own calibration method by changing CalibratePen() in PenInit.C of the device driver library.

If logo displaying is required, a Motorola logo will be displayed on the LCD screen during the pen calibration. Depending on the physical LCD screen dimensions, a different logo is displayed. There are 2 logos, one for a large screen, one for a small screen. If the LCD screen width is less than 150 pixels or the LCD height is less than 80 pixels, no logo is displayed.

Width (pixels)	Height (pixels)	Motorola Logo
width => 280	height => 150	Standard Logo
150 <= width < 280	80 <= height < 150	New Small Logo
width < 150	height < 80	None

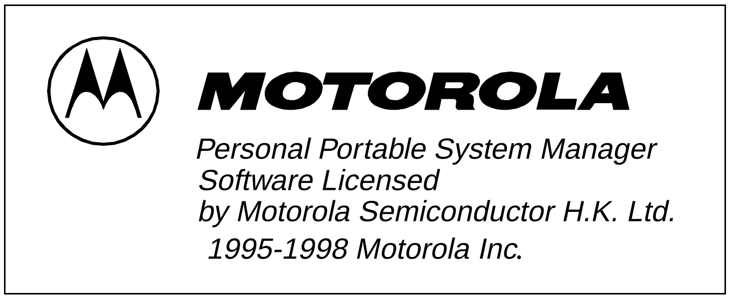

MOTOROLA
Personal Portable System Manager
Software Licensed
by Motorola Semiconductor H.K. Ltd.
1995-1998 Motorola Inc.

Figure 29-1 Standard Motorola logo

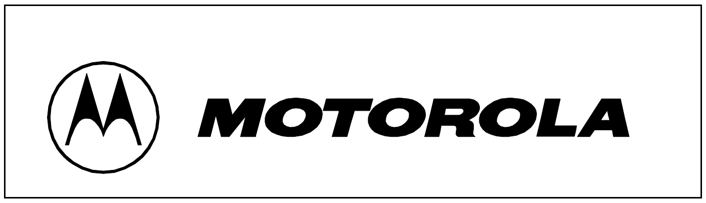


Figure 29-2 Small Motorola logo

Parameter

Name	Description
calibration	Flag to indicate if touch panel calibration is required. ` TRUE - do pen calibration ` FALSE - do not do pen calibration

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_NO_MEMORY	Not enough memory
PPSM_ERROR	Initialization failed

30.2 ReadSMVersion

Syntax

STATUS **ReadSMVersion**(P_U32 *major*, P_U32 *minor*)

Description

Returns PPSM major and minor version number. For example, for version 3.0, major number will be 3 and minor number will be 0.

Parameter

Name	Description
major	Returns the PPSM major version number
minor	Returns the PPSM minor version number

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_NO_MEMORY	Invalid input memory pointer

Chapter 31 Audio Tools

31.1 AdvAudioPlayWave (DragonBall-EZ only)

Syntax

STATUS **AdvAudioPlayWave**(P_U8 waveData, U32 waveSize, U8 prescaler,
U8 repeat, U8 clksel)

Description

This tool is valid for DragonBall-EZ only. This tool plays back a PCM audio wave signal. This is similar to the tool AudioPlayWave() but with advanced configuration details. It allows the caller to specify:

- The value of prescaler in the PWM module of DragonBall-EZ
- The repeat rate of the audio.
- The clksel in the PWM module of DragonBall-EZ

This tool assumes the user has solid knowledge of the DragonBall-EZ PWM module. For most cases, AudioPlayWave should be used.

Parameter

Name	Description
waveData	The pointer to the PCM audio wave signal
waveSize	Total number of data bytes occupied by the audio signal
prescaler	Bit 14~8 of the PWM control register, value from 0 to 127
repeat	Bit 2,3 of the PWM control register, value from 0 to 3
clksel	Bit 0,1 of the PWM control register, value from 0 to 3

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_AUDIO_REGS	Invalid values for prescaler, repeat or clksel
PPSM_ERR_AUDIO_INUSE	The PWM module is being used by another task

31.2 AudioInUse

Syntax

U8 **AudioInUse**(void)

Description

It checks if PPSM audio tools are currently being used.

Parameter

Name	Description
None	

Return Value

Name	Description
AUDIO_OFF	PPSM audio tools are not being used
WAVE_IN_USE	PPSM wave play back audio tool is being used
TONE_IN_USE	PPSM tone play back audio tool is being used

31.3 AudioPlayTone

Syntax

STATUS **AudioPlayTone**(P_U16 toneData, U32 toneSize, U16 toneDuration,
U8 autoRepeat)

Description

PPSM plays a sequence of different tone frequencies with each tone frequency having a fixed duration.

Parameter

Name	Description
toneData	The pointer to the tone frequency sequence, with frequency between 31Hz and 4048Hz
toneSize	Total number of tone frequencies to play.

Name	Description
toneDuration	The duration of each tone frequency For DragonBall-EZ ` TONE_DUR_512Hz ` TONE_DUR_256Hz ` TONE_DUR_128Hz ` TONE_DUR_64Hz ` TONE_DUR_32Hz ` TONE_DUR_16Hz ` TONE_DUR_8Hz ` TONE_DUR_4Hz For DragonBall 0 to 1000, length of duration in number of milliseconds.
autoRepeat	If auto-repeat is needed or not 0 - no autorepeat. 1 - autorepeat.

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_AUDIO_INUSE	The PWM module is being used by another task
PPSM_ERR_AUDIO_TONEDUR	Invalid tone duration The requested RTC-sampling interrupt is already being used(This only happens on the MC68EZ328)

31.4 AudioPlayWave (DragonBall-EZ only)

Syntax

STATUS **AudioPlayWave**(P_U8 waveData, U32 waveSize, U8 samplingRate)

Description

This tool is for DragonBall-EZ only. PPSM plays back a PCM audio wave signal with requested sampling rate.

Parameter

Name	Description
waveData	The pointer to the PCM audio wave signal

Name	Description
waveSize	Total number of data bytes occupied by the audio signal
samplingRate	The requested sampling rate ` SAMPLING_32KHZ ` SAMPLING_16KHZ ` SAMPLING_11KHZ ` SAMPLING_8KHZ ` SAMPLING_4KHZ

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_AUDIO_INUSE	The PWM module is being used by another task.

31.5 AudioStopTone

Syntax

STATUS **AudioStopTone**(void)

Description

Terminates the tone playing.

Parameter

Name	Description
None	

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_AUDIO_NOTINUSE	The audio tools are not in use

31.6 AudioStopWave (DragonBall-EZ only)

Syntax

STATUS **AudioStopWave**(void)

Description

Terminates the wave playing.

Parameter

Name	Description
None	

Return Value

Name	Description
PPSM_OK	Successful operation
PPSM_ERR_AUDIO_NOTINUSE	The audio tools are not in use

Part IV
System Integrator□ Guide

Chapter 32 How to make ROM?

To make PPSM applications into ROM code, two items are needed which are different from running PPSM applications using RAM memory version. These two items are:

` Boot Strap code
` Linker Specification File for ROM

This chapter gives some description of how to make a ROM code version of PPSM applications.

32.1 Boot Strap Code (boot.s)

The boot strap code performs the following functions:

` Starts the 68K core upon reset
` Map the chip-selects of MC68328 to run on a particular hardware platform
` Initialization of peripheral devices on the MC68328
` Jump into PPSM start-up code

Depending on the size and address of ROM that are used, the chip selects inside boot.s need to be changed accordingly.

32.1.1 68K Start-up

In 68K architecture, the first 256 locations in the memory address space, 0x0 to 0x400, are reserved for system vector usage and cannot be over-written with random values. The first two 32-bit words locations (address 0x00 and 0x04) are defined for the start program counter address and the stack address upon power reset.

In order to make this assignment of addresses re-locatable at link time, rather than hard-coding the addresses at compilation time, two new regions, rom_reset and rom_code, are defined by PPSM in the linker specification file to perform the mapping.

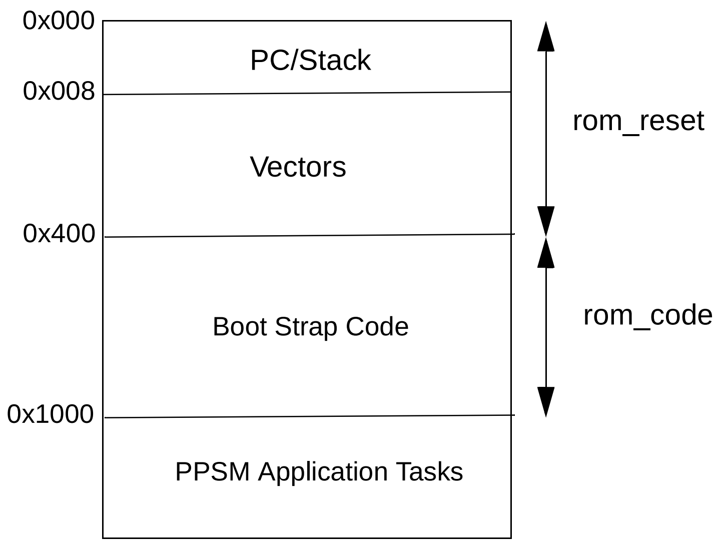


Figure 30-1 Memory map for boot strap code

32.1.1.1 ROM_RESET

This is used to map the 68K first 256 locations. In the boot strap code, it is defined as:

```
SECTION      rom_reset          ;  section declaration
DC.L         MON_STACKTOP       ;  stack address for boot code
DC.L         rom_start-ROMADDR  ;  absolute address of boot code
DCB.L        254,0              ;  interrupt vector space
```

The labels MON_STACKTOP and rom_start declared in this region are resolved with their absolute address only during link time. This implementation makes the values for these locations dynamic and system integration can be independent to the absolute location and size of the hardware system.

ROMADDR is declared in the Linker Specification File.

32.1.1.2 ROM_CODE

This regions is declared to store the boot strap code. Because this code is NOT part of PPSM library, they are declared and executed in the beginning of the memory map to avoid memory conflict.

The first line of this region MUST declare the label rom_start. This is required by the region rom_reset to work out the PC start address.

The last line of this region should be a " jmp START" instruction. This is used to start PPSM start-up code. The label START is pre-defined as the start location for the startup code.

32.1.2 Chip Selects

For the M68328ADS development board, Chip-Select group A is used for ROM and Chip-Select group B is used for RAM. Please refer to the MC68328 Integrated Processor User□ Manual, MC68328UM/AD, for details on chip select programming.

32.1.3 Peripheral Devices

Initialization of the peripherals such as default interrupt vector, watchdog and LCD controller. Please refer to the MC68328 Integrated Processor User□ Manual, MC68328UM/AD, for details on chip select programming.

32.2 Linker Supplications File for ROM

The Linker Supplications File for ROM is different to that for RAM system. The main difference being that some of the defined regions need to go into ROM address, and some regions need to go into RAM address. In general, regions that are Read-Only, such as constants, strings and code, go into ROM area; while Read/Write regions, such as ram, stack and heap space go into RAM area.

The listing below shows an example of such SPC file.

Example 30-1 Linker Specification File Example for ROM

```
partition { overlay {
      region {} rom_reset[addr=0x0];/*  reset vector in ROM */
      region {} rom_code[addr=0x400];/*  start of bootstrap code */
      region {} code[addr=0x1000];    /*  start of application code  */
      region {} const;                /*  constant data */
      region {} string;               /*  constant strings  */
      DATA = $;                       /*  pre-defined constants for
                                      initialized variables */
      LCDPHYSWIDTH = 320;             /* LCD display width */
      LCDPHYSHEIGHT = 240;            /* LCD display height */
      LCDVIRTWIDTH = 640;             /* LCD virtual width */
      LCDVIRTHEIGHT = 480;            /* LCD virtual height */
      UARTRCVBUF = 256;  /* system UART receive buffer size(in #bytes) */
} area2;
} ROM[addr=0x400000,size=0x100000];/*  1M byte ROM  */
partition { overlay {
      region {} data[addr=0x400];     /* initialized on reset */
      region {} ram[roundsize=4];     /* zeroed on reset */
      region {} malloc[size=0x80000];/* malloc space */
      region {} stack[size=0x4000];/* stack */
      STKTOP = $;                     /* SP reset value */
} area1; } RAM[addr=0x0, size=0x100000];/*  1M byte RAM  */
```

In this example, a system that has 1 MByte of ROM space mapped from address location 0x400000 and 1 MByte of RAM memory mapped from address location 0x0 has the following characteristics:

` The ROM area starts at base address 0x400000
` The region rom_reset starts from offset 0x0 from the ROM base address, which is 0x400000
` The region rom_code starts from offset 0x400 from the ROM base address, which is 0x400400

- As much executable code space in ROM as required, round to 4-byte boundary starting from 0x401000
- As much constant data space in ROM as required, round to 4-byte boundary
- As much constant strings space in ROM as required, round to 4-byte boundary
- DATA symbol to point to the downloadable address of the initialized constants to pre-initialized variables
- A LCD physical display screen of 320 pixels wide by 240 pixels high
- A panning screen of 640 pixels wide by 480 pixels high
- A 256 byte internal UART receive buffer
- The RAM area starts at base address 0x0
- As much initialized data space as required starting from an offset of 0x400, round to 4-byte boundary
- As much zeroed uninitialized data space as required, round to 4-byte boundary
- 512 KByte of heap space for dynamic memory allocation
- 128 KByte of stack space for system context switching
- A STKTOP symbol to point to the address of the 128 KByte stack

32.3 Generating S-Record File

After the PPSM application has linked with the ROM spc file, the SDS tools generates an output file in a proprietary format that is not suitable to download to ROMs.

SDS provides a tool, the loader tool, that allows the conversion from this output file into S-Record format.

32.3.1 Loader Options

To convert .OUT file into S-Record file, the following options are used:

Options		
	-d mot	generate Motorola S-Record format output file
	-o <path>\<filename>.dwn	the full name of the output file
	-m data, DATA	Copy the initial values of initialized data into ROM area
	-w <address>	Generate S-Record with offset <address> which is the base address of ROM

32.3.2 Loader Commands

Convert .OUT file format to Motorola S-Record format
down -d mot <filename>.out -m data, DATA -o <path>\<filename>.dwn -w <address>

Example 30-2 Loader command

```
down -d mot sample.out -m data, DATA -o sample.dwn -w 0x400000
```

This will convert the sample.out to S-Record format named sample.dwn which will be burned into ROM address of 0x400000.

Example 30-3 Loader command

```
down -d mot sample.out -m data, DATA -o sample.dwn
```

This will convert the sample.out to S-Record format named sample.dwn which will be burned into ROM address of 0x0.

Chapter 33 Device Drivers

PPSM supports device drivers that are either hardware dependent or third-party vendor dependent. These device drivers include:

- System configuration drivers
- Pen input driver
- LCD driver
- Handwriting recognition driver
- Font driver
- UART driver

In compilation, PIXEL_1 or PIXEL_2 needs to be defined so that the corresponding device driver library will be created.

33.1 System Configuration Drivers

There are 2 system configuration drivers:

- Boot Strap Driver
- Interrupt handler installation for the MC68328.

33.1.1 Boot Strap Driver (boot.s)

Description

The boot strap code is responsible for initialization of the MC68328 internal devices and to map chip-selects for the ROM and RAM of the hardware system at boot time. Different hardware memory configuration and system requires different boot strap code. An example boot strap code is included in PPSM device driver library to demonstrate how to boot strap and initialize the chip-selects for the M68328ADS system. Please refer to *Chapter 34 - Linker Specification File* and *Chapter 35 - Trap Usage in PPSM* for more details on hardware configuration mapping.

For hardware characteristics of the MC68328 processor, please refer to the *MC68328 Integrated Processor User Manual, MC68328UM/AD*.

33.1.2 User Interrupt Handler Installation Driver (irptdev.c)

Description

This interrupt handler device driver allows users to install their own interrupt handlers for certain kinds of interrupt.

Users can replace the default handler with their interrupt handler in the device driver to perform exception handling. Since PPSM does not manage or monitor these user-defined handlers, care must be taken when installing them. Please refer to *Chapter 15 - Interrupt Handling* and *Chapter 29 - Interrupt Handling Tools* for details on interrupt handling.

Please refer to the *MC68328 Integrated Processor User□ Manual, MC68328UM/AD*, for details on the interrupt controller.

A single argument is passed into the interrupt handler. This argument is supplied by PPSM system to specify the address of the stack pointer just before calling the user-defined interrupt handler. *Table 31-1* shows the locations of the registers relative to this stack address.

Table 31-1 Interrupt Stack Layout

	D0 - D7
	A0 - A6
	A7
	PC
Argument is the address that points to here ->	SR (16-bit)

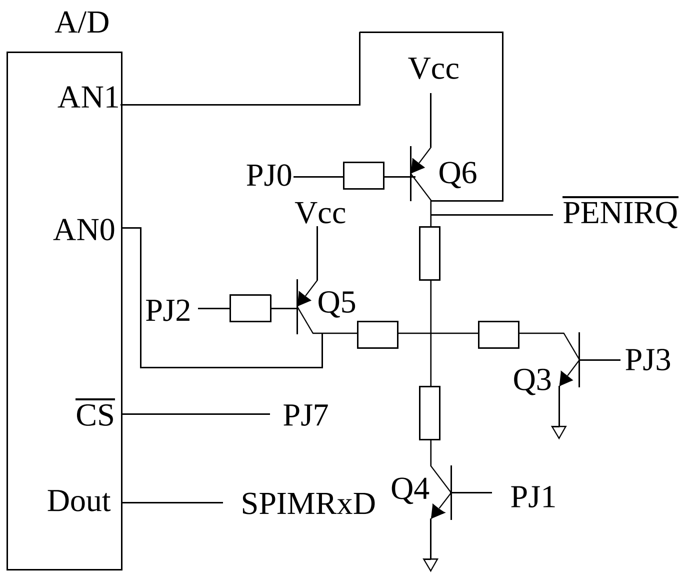
The following functions are the user defined interrupt handlers:	
void _SPIMIrptHandler(P_U32 stackPtr)	SPI Master
void _SPISIrptHandler(P_U32 stackPtr)	SPI Slave(not available in EZ)
void _IRQ6IrptHandler(P_U32 stackPtr)	IRQ6
void _UARTIrptHandler(P_U32 stackPtr)	UART
void _WatchdogIrptHandler(P_U32 stackPtr)	Watch Dog Timer
void _KeyboardIrptHandler(P_U32 stackPtr)	Keyboard
void _PWMIrptHandler(P_U32 stackPtr)	Pulse Width Modulator
void _INTIrptHandler(P_U32 stackPtr)	INT0-INT7
void _IRQ3IrptHandler(P_U32 stackPtr)	IRQ3
void _IRQ2IrptHandler(P_U32 stackPtr)	IRQ2
void _IRQ1IrptHandler(P_U32 stackPtr)	IRQ1

M68328ADS implementation:

These interrupt handlers perform no operation and return to the interrupted application immediately.

In general, _UARTIrptHandler() will not be executed. For PPSM source licensee, a "-DNO_UART_HANDLER" option can be used in compiler option to indicate that the internal UART interrupt handler is not used and this _UARTIrptHandler() is used instead.

33.2 Pen Input Device Driver (pendev.c)



X Position

PJ0	PJ1	PJ2	PJ3
0	1	1	0

Q6	Q4	Q5	Q3
On	On	Off	Off

Y Position

PJ0	PJ1	PJ2	PJ3
1	0	0	1

Q6	Q4	Q5	Q3
Off	Off	On	On

Figure 31-1 Transistor network for pen sampling

Figure 31-1 shows the configuration of the hardware used in M68328ADS for the pen input device:

` 4 I/O pins on port J, PJ0 to PJ3, are used to control the transistor network.

` I/O pin 7 on port J, PJ7, is used to control the chip-select on the A/D convertor.

` The SPI Master RxD line is connected to the A/D convertor□ digital output for the sample result.

Please refer to the *M68328ADS User□ Manual* for details on the operation of the hardware configuration.

There are 4 functions in this driver:

` Pen Initialization

` Pen Interrupt Enable

` Pen Interrupt Disable

` Pen Read Device

For M68EZ328ADS, the pen input device driver is similar as above, but it uses port D and E instead of port J in M68328ADS.

ADS version	Q3	Q4	Q5	Q6	CS
M68328ADS	PJ3	PJ1	PJ2	PJ0	PJ7
M68EZ328ADS	PD3	PD1	PD2	PD0	PE3

33.2.1 Pen Initialization

Syntax

void **PenDevInit**(void)

Description

This function initializes SPI master and all the ports that are used for pen sampling.

M68328ADS implementation:

Port J is used to control the transistor network connected to the touch panel. The default initialization values for port J are to set all pins to be output I/O pins.

Table 31-2 Port J Assignment

Port J	address	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Direction Register	0xFFF428	1	x	x	x	1	1	1	1
Select Register	0xFFF42B	1	x	x	x	1	1	1	1

For SPI Master, the control register, 0xFFF802, is set for the following:

Table 31-3 SPIM Assignment

SPIM Control Register, 0xFFF802	Bit Position	Default Value (binary)
Data Rate	15 - 13	0 1 0
SPIM Enable	9	1
Exchange Bit	8	0
SPIM Interrupt Enable	6	1
Phase Shift	5	0
Polarity	4	0
Clock Count	3 - 0	1 1 1 1

M68EZ328ADS implementation:

Port D and E are used to control the transistor network connected to the touch panel. The default initialization values for D port is to set all pins to be output I/O pins.

Table 31-4 Port D Assignment

Port D	address	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Direction Register	0xFFF418	x	x	x	x	1	1	1	1
Select Register	0xFFF419	x	x	x	x	1	1	1	1
Pullup Register	0xFFF41A	x	x	x	x	1	1	1	1
Select Register	0xFFF41B	x	x	x	x	1	1	1	1
Polarity	0xFFF41C	x	x	x	x	0	0	0	0
INT Enable	0xFFF41D	x	x	x	x	0	0	0	0
INT Edge	0xFFF41E	x	x	x	x	x	x	x	x

Port E3 is initialized to enable the A/D converter while PE0, PE1, PE2 are initialized to SPM function pins (SPMTXD, SPMRXD, SPMCLK respectively).

For SPI Master, the control register, 0xFFF802, is set for the following:

Table 31-5 SPIM Assignment

SPIM Control Register, 0xFFF802	Bit Position	Default Value (binary)
Data Rate	15 - 13	0 1 0
SPIM Enable	9	1
Exchange Bit	8	0
SPIM Interrupt Enable	6	1
Phase Shift	5	0
Polarity	4	0
Clock Count	3 - 0	1 1 1 1

33.2.2 Pen Interrupt Enable

Syntax

void **PenIrptEnable**(void)

Description

This function enables the Pen Interrupt, $\overline{\text{PENIRQ}}$, for pen down detection.

M68328ADS implementation:

Port M pin 6 is assigned as the $\overline{\text{PENIRQ}}$ pin. To enable $\overline{\text{PENIRQ}}$, the following sequence is required:

- Enable Port M pin 6 pull-up resistor
- Discharge the transistor network such that all Vcc are on, ground are off, i.e. set Port J Data register, 0xFFF439, to 0xF0
- Charge up the transistor that is connected to $\overline{\text{PENIRQ}}$, i.e. set Port J Data register, 0xFFF439, to 0x0D
- Switch Port M pin 6 to interrupt pin, i.e. set Port M Selector register, 0xFFF448, bit 6 to 0
- Enable Interrupt Mask Register, 0xFFF304, bit 20 for interrupt

Please refer to the *M68328ADS User□ Manual* for details on transistor network switching.

M68EZ328ADS implementation:

IRQ5(port F1) is assigned as the $\overline{\text{PENIRQ}}$ pin. To enable $\overline{\text{PENIRQ}}$, the following sequence is required:

- Enable Port F pin 1 pull-up resistor
- Discharge the transistor network such that all Vcc are on, ground are off, i.e. set Port D Data register, to 0xF0
- Charge up the transistor that is connected to $\overline{\text{PENIRQ}}$, i.e. set Port D Data register, to 0x0D
- Switch Port F pin 1 to interrupt pin, i.e. set Port F Selector register, bit 1 to 0
- Enable Interrupt Mask Register, bit 20 for interrupt

Please refer to the *M68EZ328ADS User□ Manual* for details on transistor network switching.

33.2.3 Pen Interrupt Disable

Syntax

void **PenIrptDisable**(void)

Description

This function disables the Pen Interrupt, $\overline{\text{PENIRQ}}$.

M68328ADS implementation:

Port M pin 6 is assigned as the $\overline{\text{PENIRQ}}$ pin. To disable $\overline{\text{PENIRQ}}$, the following sequence is required:

- Disable Interrupt Mask Register, 0xFFF304, bit 20 for interrupt
- Switch Port M pin 6 to I/O pin, i.e. set Port M Selector register, 0xFFF448, bit 6 to 1
- Disable Port M pin 6 pull-up resistor. If not switched off, it will interfere with A/D sampling

` Discharge the transistor network such that all Vcc are on, ground are off, i.e. set Port J Data register, 0xFFF439, to 0xF0
` Switch transistor network to idle mode, i.e. set Port J Data register, 0xFFF439, to 0x05

Please refer to *M68328ADS User□ Manual* for details on transistor network switching.

M68EZ328ADS implementation:

IRQ5 is assigned as the $\overline{\text{PENIRQ}}$ pin. To disable $\overline{\text{PENIRQ}}$, the following sequence is required:

` Disable Interrupt Mask Register, bit 20 for interrupt
` Switch Port F pin 1 to I/O pin, i.e. set Port F Selector register, bit 1 to 1
` Disable Port F pin 1 pull-up resistor. If not switched off, it will interfere with A/D sampling
` Discharge the transistor network such that all Vcc are on, ground are off, i.e. set Port D Data register, to 0xF0
` Switch transistor network to idle mode, i.e. set Port D Data register, to 0x05

Please refer to *M68EZ328ADS User□ Manual* for details on transistor network switching.

33.2.4 Pen Read Device

Syntax

void **PenReadDevice**(P_S16 *x*, P_S16 *y*)

Description

This function returns a reading for X and Y to the caller and switches the transistor network accordingly for reading X and Y.

M68328ADS implementation:

For X sampling, the transistors Q4 and Q6 needs to be ON while transistors Q3 and Q5 are OFF, see *Figure 31-1*, i.e. set Port J Data register, 0xFFF439, to 0xF9. While the transistors are in this setting, Port J pin 7 is asserted to activate the A/D convertor for the sampling. The digital value returned from the A/D convertor is stored in the SPIM Data register, 0xFFF800.

For Y sampling, the procedure is the same as X sampling except that the transistors Q3 and Q5 needs to be ON while transistors Q4 and Q6 are OFF, see *Figure 31-1*, i.e. set Port J Data register, 0xFFF439, to 0xF6.

The X and Y samples thus obtained are returned to the caller in the pointer arguments passed in by the caller.

M68EZ328ADS implementation:

For X sampling, the transistors Q4 and Q6 needs to be ON while transistors Q3 and Q5 are OFF, see *Figure 31-1*, i.e. set Port D Data register, to 0xF9. While the transistors are in this setting, Port E pin 3 is asserted to activate the A/D convertor for the sampling. The digital value returned from the A/D convertor is stored in the SPI Data register.

For Y sampling, the procedure is the same as X sampling except that the transistors Q3 and Q5 needs to be ON while transistors Q4 and Q6 are OFF, see *Figure 31-1*, i.e. set Port D Data register, to 0xF6.

The X and Y samples thus obtained are returned to the caller in the pointer arguments passed in by the caller.

33.3 Pen Calibration(PenInit.c)

User normally needs to do calibration once the system startup time, in order to do a correct mapping between touch panel coordination and screen display coordination. The system needs to have at least the upper-left and bottom-right corner of the touch panel coordinate in terms of the screen display coordinate to do this coordinate mapping.

Syntax

STATUS CalibratePen(U16 logoFlag)

Description

When pen calibration is necessary, PPSM calls this routine. User can replace the default pen calibration algorithm with their own. At the end of this device driver routine, the origin and maximum point of the touch panel (in terms of display screen coordinate) should be fed back to PPSM by calling PenSetInputMax(x, y) and PenSetInputOrg(x, y).

By default, the Motorola logo is displayed and two cross-hair, at the upper right and the bottom left corners, are used for pen calibration.

33.4 LCD Device Drivers (lcddev.s)

Two functions are needed in this driver for LCD controller initialization to drive the LCD panel being used in the system. Only one of the following initialization functions will be called according to the graphics mode desired.

33.4.1 1 bit/pixel Initialization

Syntax

void **_LCDDev1**(void)

Description

This function initializes the LCD controller for 1 bit/pixel graphics mode. Application programmer may add whatever statement to initialize the LCD module. Generally, LCD panel is polarity dependent which can be set in □ ove.b #$00,$21(a0)?i n lcddev.s to be adjusted for individual LCD panel.

33.4.2 2 bits/pixel Initialization

Syntax

void **_LCDDev2**(void)

Description

This function initializes the LCD controller for 2 bits/pixel graphics mode. Application programmer may add whatever statement to initialize the LCD module. Generally, LCD panel is polarity dependent which can be set in □ ove.b #$00,$21(a0)?i n lcddev.s to be adjusted for individual LCD panel.

Please refer to the *MC68328 Integrated Processor User□ Manual, MC68328UM/AD*, for details on the LCD controller and the registers definitions.

M68328ADS implementation:

_LCDDev1 initializes the following registers:

` Panel Interface Configuration Register (PICF) for a 4-bit LCD panel bus size and no gray scale
` Pixel Clock Divider Register (PXCD) to divide the clock source by 3
` Polarity Configuration Register (POLCF) to the characteristics of the LCD panel used

_LCDDev2 initializes the following registers:

` Panel Interface Configuration Register (PICF) for a 4-bit LCD panel bus size with gray scale enabled
` Pixel Clock Divider Register (PXCD) to use the system clock directly
` Gray Palette Mapping Register (GPMR) to gray scale intensity of 0x3075
` Polarity Configuration Register (POLCF) to the characteristics of the LCD panel used

33.5 Handwriting Recognition Engine Driver (hwr.c)

This driver is required for providing a common API in PPSM for recognizing handwriting input while supporting various third party handwriting recognition engines. The driver functions are needed by the Character Input Tools when handwriting recognition input is called for.

This driver consists of four functions as described in the following sections.

33.5.1 Handwriting Recognition Engine Reset

Syntax

void **ResetRecEngine**(void)

Description

This function resets the handwriting recognition engine to its default state.

This function should call a reset function (if any) provided by the handwriting recognition engine. In some cases, ResetRecEngine() may perform the same functions as InitRecEngine() (*Section 33.5.2 - Handwriting Recognition Engine Initialization*).

This function is called once when initializing the character input pad during the call of OpenInputPad().

33.5.2 Handwriting Recognition Engine Initialization

Syntax

void **InitRecEngine**(void)

Description

This function initializes or installs the handwriting recognition engine.

This function should call an initialization function (if any) provided by the handwriting recognition engine and set up data structures required by the engine for the recognition process.

This function is called once at PPSM initialization time.

33.5.3 Process One Stroke of Handwriting Input Data

Syntax

void **ProcessStroke**(U16 *numPoints*, P_POINT *strokeData*, U32 *inputAreaId*)

Description

This function processes one stroke of handwriting input data collected by the system.

Based on the input parameters, this function should convert the handwriting data for one stroke of input (if necessary) into the format being acceptable by the corresponding handwriting recognition engine. The reformatted stroke data should then be sent to the handwriting recognition engine (by calling a function provided in the handwriting recognition engine) for pre-processing or optimization before the actual handwriting recognition is initiated.

This function is called for every stroke of data collected.

Parameter

Name	Description
numPoints	the number of points in a stroke
strokeData	a pointer to a list of xy-coordinates that make up the stroke
inputAreaId	an id of the input box in which the stroke data is collected. (i.e. the identifier of the active area for the input box)

33.5.4 Initiate Character Recognition for the Handwriting Input

Syntax

void **RecognizeInput**(P_U16 *numCandidates*, P_TEXT **candidates*)

Description

This function performs the character recognition of the handwriting input.

This function should call a character recognition function provided in the handwriting recognition engine which initiates the recognition of the pre-processed stroke data collected so far. The character recognition function is expected to return to the driver function the number of character candidates being recognized, and the character codes for these candidates.

This function is called if the pen points to other character input box or there is an input time-out.

Return Value

Name	Description
numCandidates	the number of character candidates
candidates	a pointer to a list of the character codes for the candidates

Note: Since PPSM does not include a specific handwriting recognition engine, the driver functions mentioned above are default to perform no operation.

33.6 Font Driver (font.c)

This driver is required for providing a common API in PPSM for font look-up while supporting multiple font technologies and libraries supplied by various third party vendors.

The font driver consists of a data structure and two functions.

33.6.1 Font Library Information

A data structure type FONTLIB is required to store information about the font libraries being used.

The FONTLIB type is defined as follow:

```
typedef struct
      {
      P_U8  baseAddr;
      U16   fontType;
      U16   fontWidth;
      U16   fontHeight;
      U16   bitmapSize;
      } FONTLIB, *P_FONTLIB;
```

where:

1) baseAddr is the base address of the font bitmap library
2) fontType is the font type to be used for font look-up or generation
3) fontWidth is the width of the font bitmap of a character in number of pixels
4) fontHeight is the height of the font bitmap of a character in number of pixels
5) bitmapSize is the amount of memory occupied by one character font bitmap in unit of bytes

Assuming there is font bitmap or font generation engine available for each font type, the default font library information data structure could be initialized as follow:

```
FONTLIB fontLib[] =
      {
      {(P_U8)SMALL_ENG_FONT_ADDR, SMALL_NORMAL_FONT, 8, 10, 10},
      {(P_U8)SMALL_ENG_FONT_ADDR, SMALL_ITALIC_FONT, 8, 10, 10},
      {(P_U8)LARGE_ENG_FONT_ADDR, LARGE_NORMAL_FONT, 16, 20, 40},
      {(P_U8)LARGE_ENG_FONT_ADDR, LARGE_ITALIC_FONT, 16, 20, 40},
      {(P_U8)GB_FONT_ADDR, GB_NORMAL_FONT, 16, 16, 32},
      {(P_U8)BITMAP_BIG5_FONT_ADDR, BIG5_NORMAL_FONT, 16, 16, 32},
      {(P_U8)SCALABLE_BIG5_FONT_ADDR, DEFAULT_SCALABLE_FONT, 16, 16, 32}
      };
```

The fontLib data structure above are indexed into by the corresponding PPSM font types.

33.6.2 Font Library or Font Generation Engine Initialization

Syntax

void **FontInit**(void)

Description

This function initializes the font libraries and font generation engines, if applicable. This function will be called at PPSM initialization time.

Note: *Font bitmaps libraries usually do not require any initialization, whereas font generation engines do. Therefore, when applicable, this driver function should call an initialization* *routine provided by the font generation engine.*

Since PPSM does not include a specific font generation engine, this driver function is default to perform no operation.

33.6.3 Font Accessing

Syntax

P_U8 **FontGetCharAddr**(P_FONTATTR *pFont*, TEXT *code*)

Description

This function returns the font bitmap of a character based on the given font attributes and character code.

Please refer to *Section 8.4.3 - Setting Font Attributes* for the explanation of the FONTATTR data structure.

Font lookup or generation algorithms are assumed to be provided by the font supplier. This driver function should call the lookup method for the corresponding font type, to get the font bitmap of the character described by the font attributes.

Since PPSM includes 8 x 10 and 16 x 20 ASCII English fonts, the lookup method for mapping ASCII codes to English bitmap fonts are provided. The font types that use this method are:

- SMALL_NORMAL_FONT
- SMALL_ITALIC_FONT
- LARGE_NORMAL_FONT
- LARGE_ITALIC_FONT

Parameter

Name	Description
pFont	pointer to a FONTATTR structure which describes the font
code	character code for which the font lookup or generation is performed

Return Value

Name	Description
N/A	pointer to the bitmap of the character specified by the given character code (the bitmap is represented by unsigned 8-bit values

33.7 UART Device Driver (uartdev.c)

This UART device driver is a supplement to the PPSM Serial Communication Tools as described in *Chapter 26 - UART Communication Tools*. Currently this driver contains the UARTDevSendBreak() function which allows applications to send the BREAK character. This UARTDevSendBreak() function manipulates the MC68328 UART hardware registers. It is assumed that the saving, setting and restoring of appropriate system interrupt level is handled by the caller of this device driver.

33.7.1 Sending the BREAK Character

Syntax

void **UARTDevSendBreak**(U8 *sendBreak*)

Description

This function starts or stops the MC68328 UART hardware to send the BREAK character depending on the given flag.

If *sendBreak* flag is UART_SEND_BREAK, the UART hardware will start sending the BREAK character.

If *sendBreak* flag is UART_ABORT_BREAK, the UART hardware will stop sending the BREAK character.

It is assumed that *sendBreak* has to be one of the values stated. Any other value will be treated as UART_ABORT_BREAK.

Parameter

Name	Description
sendBreak	Operation flag ` UART_SEND_BREAK Starts sending BREAK characters ` UART_ABORT_BREAK Stops sending BREAK characters

Return Value

Name	Description
None	

33.8 Power Management Driver (iodev.c)

4 functions are located in iodev.c for enabling or disabling I/O ports before going to doze or sleep mode or leaving doze or sleep mode.

33.8.1 Enabling I/O ports when leaving from doze mode

Syntax

void **PortDozeEnable**(void)

Description

When PPSM wakes up from doze mode, it will call this routine to re-enable any user defined I/O ports that are not handled internally by PPSM. User must add in their own I/O initialization code in this routine.

Return Value

Name	**Description**
None	

Note: The driver function mentioned above is default to perform no operation.

33.8.2 Disabling I/O ports when going to doze mode

Syntax

void **PortDozeDisable**(void)

Description

Just before PPSM goes into doze mode, it will call this routine to disable any user defined I/O ports that are not handled internally by PPSM. User must add in the code to disable the I/O ports in this routine.

Return Value

Name	**Description**
None	

Note: The driver function mentioned above is default to perform no operation.

33.8.3 Enabling I/O ports when leaving from sleep mode

Syntax

void **PortSleepEnable**(void)

Description

When PPSM wakes up from sleep mode, it will call this routine to re-enable any user defined I/O ports that are not handled internally by PPSM. User must add in their own I/O initialization code in this routine.

Return Value

Name	Description
None	

Note: The driver function mentioned above is default to perform no operation.

33.8.4 Disabling I/O ports when going to sleep mode

Syntax

void **PortSleepDisable**(void)

Description

Just before PPSM goes into sleep mode, it will call this routine to disable any user defined I/O ports that are not handled internally by PPSM. User must add in the code to disable the I/O ports in this routine.

Return Value

Name	Description
None	

Note: The driver function mentioned above is default to perform no operation.

Chapter 34 Linker Specification File

PPSM makes use of the Linker Specification File (.SPC file) of the SDS development environment for memory mapping and configuration. This SPC file defines the locations and contents of memory regions and allow user-defined symbols. It is written in a C-like language. For full details on linker specification files, please refer to the SDS User Manual.

PPSM required a few user-defined symbols for it to operate. They are listed in *Table 32-1*.

Table 32-1 Symbols Definitions for Linker Specification File

Symbol Name	Description
LCDPHYSWIDTH	LCD physical display screen width in pixels
LCDPHYSHEIGHT	LCD physical display screen height in pixels
LCDVIRTWIDTH	LCD virtual screen (Panning Screen) width in pixels
LCDVIRTHEIGHT	LCD virtual screen (Panning Screen) height in pixels
UARTRCVBUF	Internal UART receive buffer size in number of bytes

System integrators inform PPSM of the physical memory size available in a given hardware system using this linker specification file.

PPSM maintains a stack and heap for memory usage in the application program. The system integrator must decide how much memory is to be given for the stack and heap. In general, the heap memory is used as the dynamic allocatable memory for application use, and the stack memory is mostly used by the system. For details on memory mapping, please refer to *Chapter 32 - How to make ROM?*.

PPSM also maintains an internal UART receive buffer which is used as temporary storage for data received from the UART. The system integrator must decide what is the optimal buffer size for a given hardware system.

34.1 .SPC File for a RAM-only System

Following is an example of a SPC file for a M68328ADS with debug monitor ROM (for SDS use only) with 2 Mbyte of RAM for system use. This 2 MByte RAM-only system has the following characteristics:

- The RAM system starts at address 0x0
- 4 KByte of reserved memory starting from address 0x0 for reset vectors and SDS debug monitor use
- As much executable code space as required, round to 4-byte boundary
- As much constant data space as required, round to 4-byte boundary

- As much constant strings space as required, round to 4-byte boundary
- As much initialized data space as required upon reset, round to 4-byte boundary
- As much zeroed memory space as required, round to 4-byte boundary
- 704 KByte of heap space for dynamic memory allocation
- 128 KByte of stack space for system context switching
- A STKTOP symbol to point to the top of the 128 KByte stack
- a DATA symbol to point to the downloadable address
- A LCD physical display screen of 320 pixels wide by 240 pixels high
- A panning screen of 640 pixels wide by 480 pixels high
- A 256 byte internal UART receive buffer
- 2 MByte of RAM

Example 32-1 Linker Specification File Example

```
partition { overlay {
      region {} reset[addr=0,size=0x1000];/* reset vector */
      region {} code[roundsize=4];/* executable code */
      region {} const[roundsize=4];/* constant data */
      region {} string[roundsize=4];/* constant strings */
      region {} data[roundsize=4];/* initialized on reset */
      region {} ram[roundsize=4];/* zeroed on reset */
      region {} malloc[size=0xB0000];/* malloc space */
      region {} stack[size=0x20000];/* stack */
      STKTOP = $;/* SP reset value */
      DATA = $;/* ata?download addr */
      LCDPHYSWIDTH = 320;/* LCD display width */
      LCDPHYSHEIGHT = 240;/* LCD display height */
      LCDVIRTWIDTH = 640;/* LCD virtual width */
      LCDVIRTHEIGHT = 480;/* LCD virtual height */
      UARTRCVBUF = 256;              /* system UART receive buffer size(in
      #bytes) */

} example; } RAM[addr=0x0, size=0x200000]
```

34.2 .SPC File for a ROM-RAM System

The .SPC file for a ROM-RAM system consists of two partitions. One partition for the ROM layout and one partition for the RAM layout on the system. Please refer to *Chapter 32 - How to make ROM?* for a specific example of a .SPC file for such a system.

34.3 For SingleStep Debugging System (SDS) user

One of the methods to find out the optimum size specified in the malloc field in the .SPC file will be shown below. This method may not be applicable to other debugging system.

1) First of all, we set the RAM size to 2 M, so as the malloc size. For example,

```
partition { overlay {
    region {} reset[addr=0x0];                  /* reset vector */
.....
```

```
    region {} malloc[size=0x200000];             /* malloc space */
......
} example; } RAM[addr=0x400000, size=0x200000];
```

2) Following error messages will be generated after linking:

```
LINKER: error: actual ’RAM’ size (0x21F762) is larger than specified size
LINKER: error: overlay ’example’ is larger than the partition containing it
NMAKE : fatal error U1077: ’LINKER’ : return code ’0x1’
Stop.
```

3) Then, we can calculate the optimum malloc size by subtracting exceeding bytes from the total memory size:

```
                   0x200000 - 0x1F762 = 0x1e089e
```

4) Finally, we can write the optimum size in the malloc field:

```
partition { overlay {
    region {} reset[addr=0x0];                   /* reset vector */
.....
    region {} malloc[size=0x1e089e];             /* malloc space */
......
} example; } RAM[addr=0x400000, size=0x200000];
```

5) This optimum malloc size needed to be found again after modified source code by similar method. This size is recommended to be found after debugging stage.

Chapter 35 Trap Usage in PPSM

PPSM tools access the PPSM system kernel internally via the TRAP instruction. When an application makes a PPSM toolset call, a pre-defined TRAP instruction is called with the correct arguments assigned to the various registers instead of calling the actual function.

On the 68K, there are 16 TRAPs that can be used. *Table 34-2* shows the TRAP instruction mapping for the PPSM tools.

Table 34-2 Trap Instructions Mapping

TRAP Number	TRAP Vector	TRAP Address	PPSM Tools
1	0x21	0x84	PPSM Basic Tools
2	0x22	0x88	PPSM Extended tools
3	0x23	0x8C	Touch Panel tools
4	0x24	0x90	LCD Display tools
5	0x25	0x94	Timer tools
6	0x26	0x98	PCMCIA IC Card
7	0x27	0x9C	Communication tools

35.1 PPSM Tools Calling Structure

For the protection of the PPSM object code and security, two levels of indirection is introduced in the PPSM when an application makes a system call.

The first level is a stub library call. A PPSM tools stub interface is supplied to the application developers. This is a complete library of the PPSM tools but the function body consists of parameter passing and TRAP instruction call only.

The second level is the TRAP system call, which takes in the parameters passed in by the stub library and makes an actual system call to the PPSM function.

With this implementation, application developed code can reside in RAM, while PPSM system code can be in ROM. Hence, ensuring object code protection and portability of the PPSM system code.

35.2 TRAP Implementation

Figure 34-1 shows a block diagram of the event that takes place when an application makes a system call. The stub library and the TRAP instruction jump sections are implemented in low level assembler language to minimize the overhead induces during a system call.

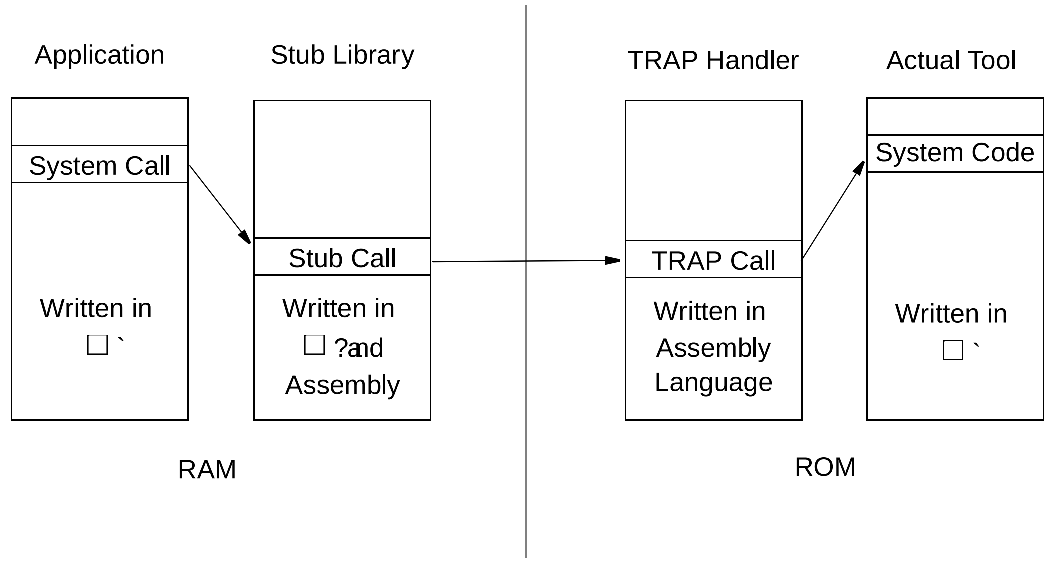


Figure 34-1 PPSM Tools Calling Structure

35.2.1 Application

Applications are written in ANSI C. All PPSM system tools use ANSI C procedure calling convention. Because SingleStep C compiler allows assembly language in C source code, it is also allowed in PPSM applications.

35.2.2 Stub Library

The Stub Library provides an interface between application code and system routines. Its main function is for parameter passing and calling TRAP instruction with the correct arguments. It is written in a ANSI C with 68K assembly language embedded format. Each function in the stub library has the general layout as shown below:

```
int    simplification( int a, int b)
       #define NO_OF_ARGUMENT2
       #define FUNC_NUMBER5
       #define TRAP_NUMBER3

       int    rv;    /*  return value  */
       asm(
              `      LEA    8(A6),A0?
              `      MOVE.W #NO_OF_ARGUMENT,D1?
              `      MOVE.W #FUNC_NUMBER,D0?
              `      TRAP   #TRAP_NUMBER?
              `      MOVE.L D0,{rv}?
       );
       return (rv);
       }
```

The first instruction that SingleStep C compiler generates for the start of a □ ? procedure is a □INK?instruction. This instruction sets the frame pointer, A6, to points to the stack that is used by the procedure caller. To access the first argument, an offset of 8 is added to A6, see *Figure 34-2*. The Stub Library passes the caller□ argument list to the TRAP instruction via the address register A0 and the PPSM tools function number via D0.

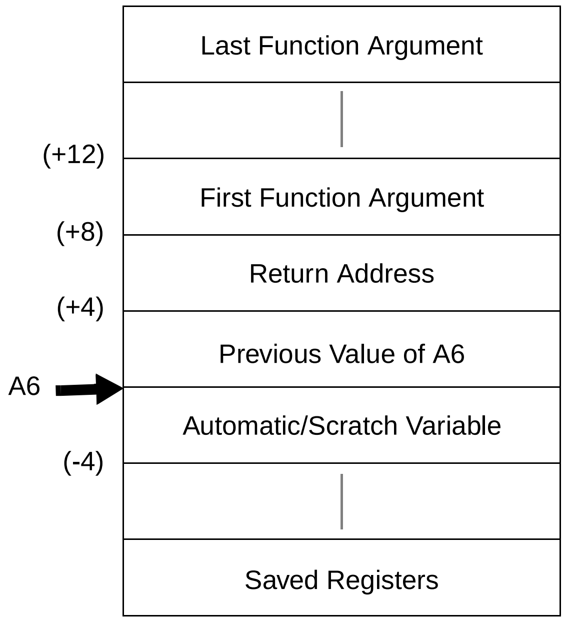


Figure 34-2 Frame Pointer Location in Stack Layout

35.2.3 TRAP Service Handler

The TRAP Service Handler routines are implemented in 68K assembly language. There are 7 jump tables, one for each TRAP service handler (See *Table 34-2*). The jump table consists of an array of PPSM tool addresses. These addresses are used as the PPSM system function procedure addresses by the TRAP service handler when it receives a call from the Stub Library. The listing below shows the general code layout for a TRAP service handler.

```
.FREF  _Func_1,2
.FREF  _Func_2,8

JMPTABLE:
DC.L   _Func_1, _Func_2

Trap_1:
LSL.W  #2,D1  ; Multiply no of arg by 4
ADD.L  D1,A0  ; Move A0 to end of arg list
LSR.W  #2,D1  ; Move D1 back to no of arg
loop_1:
TST.W  D1     ; Any parameter left
BEQ    loop_2 ; If not, goto loop_2
MOVE.L -(A0),-(A7); Move parameter to stack
SUBQ.W #1,D1  ; Decrement D1
BRA    loop_1 ;
loop_2:
LEA    JMPTABLE,A0; Get jump table address
LSL    #2,D0  ; Get function offset
ADD.L  D0,A0  ; Find function address
MOVEA.L(A0),A1; Move address to A1
JSR    (A1)   ; Jump to system routine
RTE
```

Appendices

Appendix A Error Code Definition

The following are the definitions of all the possible error codes:

Table A-1 Error Code Definition

Error Code	Definition
PPSM_OK	Message successfully sent
	Successful operation
	Task successfully terminated
PPSM_ERROR	Active list stack empty
	Initialization failed
	numberOfPoints is 0
	Soft keyboard is not opened
	Unsuccessful operation
PPSM_ERR_ACTIVE_POP	Active list stack empty
PPSM_ERR_ACTIVE_PUSH	Unable to push active list
PPSM_ERR_AREA_CODE	Invalid area code for active area type
PPSM_ERR_AREA_ID	Invalid active area identifier
	Invalid active area pointer
PPSM_ERR_AUDIO_INUSE	The PWM module is being used by another task
PPSM_ERR_AUDIO_NOTINUSE	The audio tools are not in use
PPSM_ERR_AUDIO_REGS	Invalid values for prescaler, repeat or clksel
PPSM_ERR_AUDIO_SAM	Invalid audio sampling rate
PPSM_ERR_AUDIO_TONEDUR	Invalid tone duration
PPSM_ERR_BAUD	Invalid baud rate flag
PPSM_ERR_BUSY	There is already an on-going receive request (when trying to initiate one)
PPSM_ERR_CHARLEN	Invalid character length flag
PPSM_ERR_COLOUR	Invalid pen color
PPSM_ERR_COORDINATE	Invalid coordinates
PPSM_ERR_CURSOR_INIT	Invalid cursor pointer

Table A-1 Error Code Definition

Error Code	Definition
PPSM_ERR_DAY	Invalid day pointer
	Invalid day value
PPSM_ERR_DB_ADD	Unable to add database
PPSM_ERR_DB_DBID	Invalid database identifier
PPSM_ERR_DB_FDID	Invalid field identifier
PPSM_ERR_DB_READNO	Invalid number found
PPSM_ERR_DB_RECID	Invalid record identifier
PPSM_ERR_DB_SFLAG	Invalid secret flag value
PPSM_ERR_DB_TYPE	Invalid data type
PPSM_ERR_DOT_WIDTH	newWidth is 0
PPSM_ERR_DOZE_TIME	Doze time-out period out of range
PPSM_ERR_DRAW_INIT	Invalid draw pointer
PPSM_ERR_FILL_PATTERN	Invalid fill pattern mode
PPSM_ERR_FILL_SPACE	Invalid space gap in fill pattern
PPSM_ERR_GREY	Invalid grey level value
PPSM_ERR_HEIGHT	Invalid graphics output area height
	Invalid height
PPSM_ERR_HOUR	Invalid hour pointer
	Invalid hour value
PPSM_ERR_ICON_TYPE	Invalid predefined icon type for control icon
PPSM_ERR_INPUT_PAD_CLOSED	Input pad is not opened
PPSM_ERR_INPUT_PAD_HEIGHT	Input pad height out of range
PPSM_ERR_INPUT_PAD_NOSCREEN	No panning screen for input pad
PPSM_ERR_INPUT_PAD_OPENED	Input pad already opened
PPSM_ERR_INPUT_PAD_WIDTH	Input pad width out of range
PPSM_ERR_INPUT_PAD_X_POS	Input pad x-coordinate out of range
PPSM_ERR_INPUT_PAD_Y_POS	Input pad y-coordinate out of range
PPSM_ERR_INVALID_ACCESS	Invalid access of UART
PPSM_ERR_INVALID_TMOUT	Time-out value out of range
PPSM_ERR_IRPT_HDLER	Handler not requested by application
PPSM_ERR_LCD_FONT	Invalid font type

Table A-1 Error Code Definition

Error Code	Definition
PPSM_ERR_LCD_GREY	Invalid background grey level
	Invalid grey level value
PPSM_ERR_LCD_RADIUS	Invalid radius
PPSM_ERR_LCD_STYLE	Invalid style
PPSM_ERR_LCD_X	Invalid x-coordinate
PPSM_ERR_LCD_Y	Invalid y-coordinate
PPSM_ERR_MINUTE	Invalid minute pointer
	Invalid minute value
PPSM_ERR_MODE	Invalid operating mode flag
PPSM_ERR_MONTH	Invalid month pointer
	Invalid month value
PPSM_ERR_NO_ALARM	No alarm is set
PPSM_ERR_NO_MEMORY	Not enough memory
PPSM_ERR_NO_REQUEST	Receive request was not initiated
PPSM_ERR_NUM_FMT	Invalid user format field number
PPSM_ERR_PAN_ADDRESS	Invalid panning screen address
PPSM_ERR_PAN_HEIGHT	Invalid panning screen height
PPSM_ERR_PAN_INIT	No panning screen has been created
PPSM_ERR_PAN_WIDTH	Invalid panning screen width
PPSM_ERR_PARITY	Invalid parity flag
PPSM_ERR_PEN_GET	No active area has been created
PPSM_ERR_PEN_INIT	No touch panel exist
PPSM_ERR_PEN_RATE	Invalid sampling period specified
PPSM_ERR_PERIOD	Invalid time-out period
PPSM_ERR_RCTS_IDLE	RTS/CTS flow control is not enabled
PPSM_ERR_REF_MAX	Alarm time is more than 1 minute from current time
PPSM_ERR_RELEASE	Unable to release handler
PPSM_ERR_SECOND	Invalid second pointer
	Invalid second value
PPSM_ERR_SKBD_USED	Soft keyboard already being used
PPSM_ERR_SKBD_XSIZE	Soft keyboard width out of range

Table A-1 Error Code Definition

Error Code	Definition
PPSM_ERR_SKBD_X_POS	Soft keyboard x-coordinate out of range
PPSM_ERR_SKBD_YSIZE	Soft keyboard height out of range
PPSM_ERR_SKBD_Y_POS	Soft keyboard y-coordinate out of range
PPSM_ERR_SLEEP_TIME	Sleep timeout period out of range
PPSM_ERR_STOPBIT	Invalid number of stop bits flag
PPSM_ERR_TASK_FLAG	Invalid screen flag
PPSM_ERR_TASK_HEIGHT	Invalid screen height
PPSM_ERR_TASK_ID	Invalid address for storing task identifier
	Invalid task identifier
PPSM_ERR_TASK_WIDTH	Invalid screen width
PPSM_ERR_TEXT_CR	Error while creating text template
PPSM_ERR_TEXT_CUR	Invalid character cursor position
PPSM_ERR_TEXT_FONT	Invalid font type
PPSM_ERR_TEXT_GREY	Invalid text grey level value
PPSM_ERR_TEXT_HEIGHT	Given height extends text display area beyond the panning screen
PPSM_ERR_TEXT_ID	Invalid text template identifier
PPSM_ERR_TEXT_STYLE	Invalid output style type
PPSM_ERR_TEXT_WIDTH	Given width extends text display area beyond the panning screen
PPSM_ERR_TEXT_X	Text template x-coordinate out of range
PPSM_ERR_TEXT_Y	Text template y-coordinate out of range
PPSM_ERR_TMOUT_VALUE	Invalid time-out period
PPSM_ERR_WIDTH	Invalid graphics output area width
	Invalid width
PPSM_ERR_X_POS	Invalid dot x position
PPSM_ERR_YEAR	Invalid year pointer
	Invalid year value
PPSM_ERR_Y_POS	Invalid dot y position
PPSM_NO_MATCH	No match of data

Appendix B List of References

1) *Addendum and Errata to MC68328 DragonBall^{TM} Integrated Microprocessor User□ Manual*. Motorola, Inc.

2) *Computer Graphics Principles and Practices*. Foley, Van Dam, Feiner and Huges. Addison Wesley, 1993.

3) *CrossCode C for the 68000 Microprocessor Family*. Software Development Systems, Inc.

4) *MC68328ADS Application System User□ Manual*. Motorola, Inc.

5) *MC68328 Integrated Processor User□ Manua*l, MC68328UM/AD.Motorola, Inc.

6) *SingleStep^{TM} Debugger for the 68000 Microprocessor Family*. Software Development Systems, Inc.

7) *M68EZ328ADS Application Development System User□ Manual*. Motorola, Inc.

8) *MC68EZ328 Integrated Processor User□ Manual*. Motorola Inc.

Appendix C Index of PPSM Tools