# PPSM-GT

Personal Portable System Manager

# USER GUIDE

version 1.1 dated 3/12/2001

MOTOROLA

For more information on PPSM-GT, please contact Motorola Semiconductors Hong Kong Ltd by calling 852-2666-8333.

## How to Contact Motorola:

| | |
|---|---|
| **Hong Kong (A/P Headquarters)** | Motorola Semiconductors Hong Kong Ltd.<br>Silicon Harbour Center<br>2 Dai King Street<br>Tai Po Industrial Estate<br>Tai Po, New Territories<br>Hong Kong<br>Tel: 852 2666 8333<br>Fax: 852 2666 6123 |
| **World Wide Web** | `http://www.ppsmgt.com` |
| **Sales & Technical Support** | Motorola Semiconductors Hong Kong Ltd.<br>Room 2307-2312, 23 / F<br>Metro Plaza Tower II<br>223 Hing Fong Road, Kwai Chung<br>New Territories<br>Hong Kong<br>Tel: 852 2489 1111<br>Fax: 852 2480 5437 |

# About This Book

This user guide describes the features and APIs of Personal Portable System Manager GT (PPSM-GT™). It explains in detail how to develop application program using the PPSM-GT operating system and the DragonBall™ family of products and other microprocessors.

## Audience

This user guide is intended to assist developers with using PPSM-GT in a wide variety of products. The document presumes basic knowledge of the following:

- Features of the Motorola MC68VZ328 (DragonBall VZ)
- 68000 assembly language programming
- The Metrowerks CodeWarrior integrated development environment

## Organization

The *PPSM-GT User Guide* is organized into sections on the architecture and programming of the PPSM_GT operating system. Each section contains multiple chapters. Summaries of the chapters, subdivided by section, follow.

Chapter 1, "Introduction" contains an overview of PPSM-GT, including its design architecture and services provided.

### "Background"

Chapter 2, "PPSM-GT Core Fundamentals" provides programming information about the operating system's basic building blocks.

Chapter 3, "PPSM-GT Core Programming Concepts" describes the basic programming concepts for programming with the PPSM-GT core. It includes a description of the PPSM-GT main program.

## "Getting Started"

Chapter 4, "Installing PPSM-GT" provides information for installing PPSM-GT and defines the hardware and software requirements for the PPSM-GT environment.

Chapter 5, "Developing a New Application" provides information about configuring a new project in the PPSM-GT environment and a CodeWarrior project for a PPSM-GT application.

Chapter 7, "ISR Routines Services" provides information about how to handle PPSM-GT ISR routine.

## "Developing with System Services"

Chapter 8, "Kernel Services" describes the services that provide access to the nerve center of PPSM-GT. The kernel consists of an embedded pre-empted operating system for multitasking applications. It is responsible for task manipulation and, together with power and memory management, commands the proper operation of the system.

Chapter 9,  "Memory Management Services" describes the services that control the distribution of the system memory resources.

Chapter 10,  "Power Management Services" describes the services that handle operating the system to achieve a balance between performance and power efficiency.

Chapter 11,  "System Application Services" describes the services that form the structure of the whole program. This structure is the foundation and usually used in the main program.

Chapter 12, "System Event Management Services" describes the services that contribute to the interworking of the system.

Chapter 13, , "Software Timer Handling Services" describes the services that provide timers for handling the timing of the whole system.

## "Developing with Application Services"

Each of the chapters in this section describes the services that are identified in its title.

## "Graphics & Input Handling Services"

Each of the chapters in this section describes the services that are identified in its title.

## Suggested Reading

### Printed Resources

A brief sampling of titles includes the following:

*CodeWarrior IDE User Guide* and *Debugger User Guide*.

### Networking

*TCP/IP Illustrated Volume 1: The Protocols* by W. Richard Stevens (ISBN 0-201-63346-9)

*TCP/IP Illustrated Volume 2: The Implementation* by Gary R. Wright and W. Richard Stevens (ISBN 0-201-63354-X)

*Internetworking with TCP/IP Volume 1: Principles, Protocols, and Architecture*, Second Edition, by Douglas E. Comer (ISBN 0-13-468505-9)

*Internetworking with TCP/IP Volume 2: Design, Implementation, and Internals*, Second Edition, by Douglas E. Comer (ISBN 0-13-125527-4)

*Troubleshooting TCP/IP: Analyzing the Protocols of the Internet* by Mark A. Miller P.E. (ISBN 1-55851-268-3)

*The Simple Book: An Introduction to Internet Management*, Second Edition, by Marshall T. Rose (ISBN 0-13-177254-6)

*UNIX Network Programming* by W. Richard Stevens (ISBN 0-13-949876-1)

## Online Resources

There are many Web sites dedicated to C++ and object-oriented programming. Some excellent examples are:

- `http://www.codewarrioru.com/CodeWarriorU/`
- `http://www.cerfnet.com/~mpcline/c++-faq-lite/`
- `http://www.research.att.com/~bs/C++.html`

## Conventions

This guide uses the following conventions:

- Code examples, file names, and universal resource locators (URLs) are set in a monospace font. For example, a variable declaration is presented as follows: `int myInteger`;

- Menu names, menu items, and buttons are shown in **bold** text. For example, “Choose **File > Delete** to remove the file.”
- Book names, section names, chapter names, and key words are presented in *italic* text. For example, “Refer to the *PPSM-GT API Reference* for more information.”

## Definitions, Acronyms, and Abbreviations

See the chapters in the Background section for definitions of various fundamental concepts and terms.

The following abbreviations are used in this document:

| | |
|---|---|
| API | application program interface |
| GUI | graphical user interface |
| ISR | interrupt service routine |
| IrDA | Infrared Data Association |
| PDA | personal digital assistant |
| RTOS | real-time operating system |
| RTC | real-time clock |
| SCI | serial communication interface |
| TCP/IP | transmission control/internet protocol |
| UART | universal asynchronous receiver/transmitter |

# 1 Introduction

Welcome to the world of Personal Portable System Manager GT (PPSM-GT™).

## What Is PPSM-GT?

Personal Portable System Manager GT is a compact operating system for handheld smart products and other LCD-display products. It is designed specifically for devices that use the Motorola DragonBall™ family of microprocessors. This operating system enables handheld electronic products with LCD displays such as advanced pagers, advanced cellular phones, game machines, GPS instruments, organizers, and personal digital assistants (PDAs).

PPSM-GT is a real-time, 32-bit multitasking kernel with prioritized interrupt scheduling. All tasks are prioritized and grouped as either real-time tasks or time-sliced tasks. Time-sliced tasks are real-time tasks that have the same priority level.

PPSM-GT is written in C, and the current version is designed to work on Dragonball VZ processor.

PPSM-GT is not just a kernel; it is a system developer toolkit. It consists of pen input, graphics, audio, RTC, alarm, text management, character input, software keyboard, power management, system, and communications services. Application developers can design a sophisticated user interface and configure a DragonBall processor (or other microprocessor) with easy-to-use APIs for LCD-based products. The PPSM-GT toolset, together with its device drivers, provides the basic control of the LCD, the drawing functions, the real-time clock, and the UART, among other components and functions.

The PPSM-GT 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, PPSM-GT gives system integrators greater flexibility to use various types of hardware devices without changing the core of the software.

## Why Use a kernel like PPSM-GT?

There are many benefits associated with the use of a real-time kernel, including the reduction of software development time and cost. Because the kernel serves as a foundation, the rules provided by the kernel make it easier to develop application code. This ease improves programmer productivity. However, this benefit can be lost with a custom-designed kernel. An application developer using such a kernel would have to produce the operating system code as well as the application code. A commercial real-time operating system like PPSM-GT permits the developer to focus all efforts on the application, reducing total development time.

Equally important are the benefits of enhanced product reliability, maintainability, and quality. In contrast to an untested custom kernel, a commercial operating system provides proven services and code to meet application needs such as that found in the GT2 services. In addition, such a system is better documented than custom kernels are with regard to design policies, rules, and kernel services, which improves application code maintenance.

## PPSM-GT Architecture

**Figure 1.1 PPSM-GT Design Architecture**

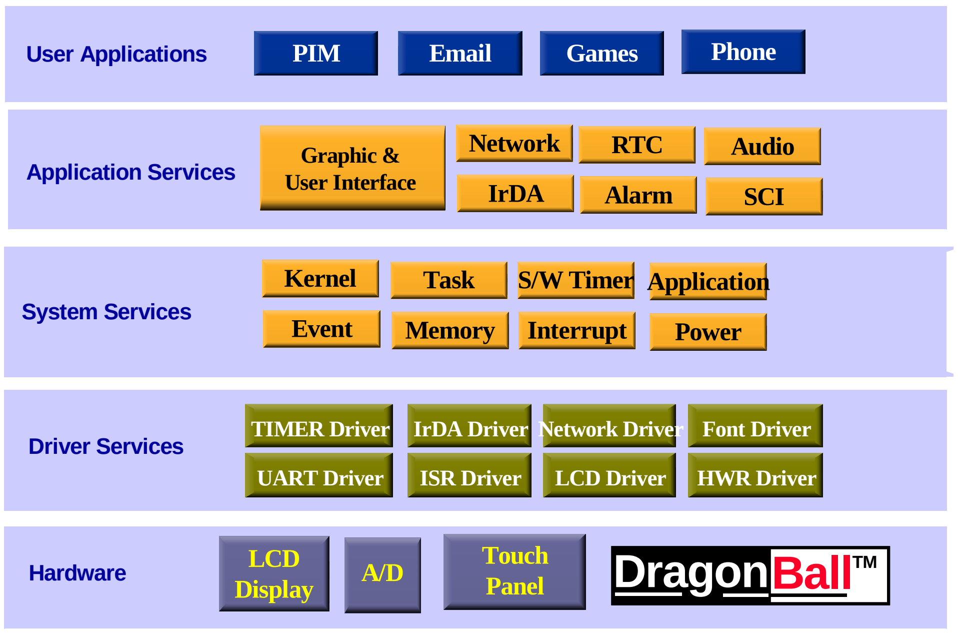


Figure 1.1 depicts the operating system's design architecture.

## System Core Services

The system core services consists of the kernel, power management, memory management and system application services. This group acts as the control center of PPSM-GT. It composes an embedded pre-empted operating system for multitasking applications. The kernel is responsible for task manipulation and, together with power and memory management, commands the proper operation of the system. Power management handles the status of system operation to achieve a balance between performance and power efficiency. System application setup the operating environment for the applications and are normally used in the main program.

Another group of system core services contributes to the interworking of the system. These services include event management, software timer handling, interrupt service routine services, and device driver services.

## Kernel Services

The kernel is a software component and is not considered a part of the application, and has the function to determine what application program is to gain control of the CPU. Kernel executive manages the system resources and also provides an architectural framework for tasks, semaphore and event-driven operation etc to be put together in application routines so as to achieve desired operational behaviors of devices. Kernel executive creates tasks that are a series of routines that can execute concurrently to implement the application design, and allows priority levels be defined on tasks to define the relative importance of tasks. Kernel Executive also supports semaphore to control access of “critical region” in shared resources and event-driven operation to allow the kernel to respond predictably to events as they occur.

## Memory Management Services

Memory management Services provide APIs to enable applications to access local memory space. PPSM-GT manages a heap that allows callers to dynamically allocate memory from the system. When PPSM-GT is being used, the standard memory tools that are provided by the compiler are disabled.

PPSM-GT also provides a set of memory allocation and inquiry tools to enable applications to get the run-time memory size.

## Power Management Services

PPSM-GT utilizes the power control module of DragonBall to implement a set of power management tools to conserve system power. Applications can control the system’s power management features directly or use PPSM-GT’s automatic power management features.

### Direct Control

A set of tools provides applications with the ability to directly control the following in normal mode:

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

### Automatic Control

A set of tools are available for the caller to set the parameters for automatic power-management features, including the following:

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

## System Application Services

System application services are usually used to setup the operating environment for the application. APIs in this services setup the display as well as the platform for tasks to operate. They are like the foundation in a building project, and normally used in the start of the program.

## Event Management Services

PPSM-GT provides event management services to handle events that are generated by a user, task, or device driver. Events are buffers that are sent from one task to another. They contain information that is sent to a target task from another task or a device driver.

There are two groups of events in PPSM-GT: normal and broadcast events. Each group can be further divided into the following four types:

- Erasable wake-up events
- Erasable non-wake-up events
- Non-erasable wake-up events
- Non-erasable non-wake-up events

## Software Timer Handling Services

The software timer functions like an alarm, and it takes the software reference timer as its reference. The resolution of the software timer is in milliseconds. The range of the software timer is half that of the reference timer: 0 to 2,147,500 seconds, or 0 to 24.85 days. PPSM-GT provides APIs to control, set, and use the software timer.

### Interrupt Service Routine (ISR) Services

ISR services provide application interfaces to the DragonBall interrupt controller. The interrupt software request API is used as a device driver interface between the application and the hardware. It configures the drivers that are built on the top of DragonBall's interrupt controller.

### Device Driver Services

PPSM-GT supports variant target hardware configurations and third-party fonts. This support is provided through the modification of PPSM-GT device drivers, including the following drivers:

- Font driver
- LCD driver
- Pen input driver
- System boot-up driver
- System interrupt handling drivers
- System power control driver

## Application Services

Application services is a group of services that provided the features of the system and the communication between the system and external devices. The services include real-time clock handling, alarm services, pen input handling, software keyboard services, touch screen input pad services, text management, graphics management, application download services, serial communication interface services, audio management, IrDA services, and TCP/IP services.

### Real-Time Clock Handling Services

PPSM-GT real-time clock handling services are provided to handle DragonBall microprocessors' RTC modules. The APIs enable ease of use, including checking real-time clock information.

## Alarm Services

PPSM-GT alarm services are used for handling DragonBall microprocessors’ RTC alarm. The APIs enable ease of use and support multiple one-shot or periodic alarms.

## Application Download Services

Application download services enable the system integrator or software developer to provide an area of the software for downloaded applications. Third-party software developers can write their own software and download it into this area of the system for execution.

In PPSM-GT, a downloaded application is called an application image to distinguish it from the system resident application.

PPSM-GT provides APIs to convert a downloaded application image into a system application and to delete a downloaded application, freeing the memory allocated for it.

## Audio Management Services

PPSM-GT supports three types of audio: tone, wave, and melody. 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 being used by another task or application.
- When audio playing finishes, an event that indicates this completion is sent to the task that called audio services.

## Serial Communication Interface Services

PPSM-GT supports multiple serial communications through the serial communication interfaces (SCIs) in both normal mode and IrDA mode. The exact number of SCI resources that are supported is limited by hardware (refer to the appropriate hardware manual for details). When the common set of SCI services to send and receive data through the SCIs is used, each resource has an identifier that distinguishes it from other resources.

Each SCI resource also has an internal receive buffer. The default size is specified in the included header file. This size can be dynamically changed during run time.

By default, SCI resources are disabled. They should be enabled before use.

### IrDA Services

The IrDA management services in PPSM-GT are a layered set of protocols particularly aimed at point-to-point infrared communications and the applications needed in that environment. Two protocols are supported:

- IrCOMM services are IrDA services that were designed to provide serial and parallel port emulation to legacy applications. These services enable the applications to communicate with a peer device over an IrDA infrared link instead of the wired link.
- IrOBEX services are an implementation of the IrDA Object Exchange specification (IrOBEX) for IrDA protocol stacks. They provide the ability to “Put” data objects very simply and flexibly, thereby enabling rapid application development and interaction with a broad class of devices including PCs, PDAs, data collectors, and cameras.

### Networking Services

Networking services support a set of protocols to allow cooperating computers or devices to share resources across a network. Sockets are endpoints of communications and a sockets API similar to the Berkley socket protocol is provided. They provide an interface to communications protocol for common operations. These include sending data, receiving data, establishing connections and configuring networks.

In addition to socket, the networking services also provide protocols such as PPP, and TCP/IP for doing specific tasks such as transferring files between computers, remote logins and sending mail.

## Graphics & User Interface Services

Graphics and User Interface services provides basic features such as displaying of images and text, drawing of lines and shapes, handling of input pads and software keyboards.

## Graphic Manipulation Services

The Graphics manipulation services provided the following main functions:

- Getting and setting of display parameters
- Drawing lines and shapes
- Displaying and manipulating of bitmap images
- Control hardware cursor

## Text Management Services

Text Management supports setting up of templates to display 8-bit and 16-bit text data representation. This 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.

## Software Keyboard Services

Software keyboard services provide a soft keyboard for applications to receive character inputs from the user. There is a default version and customized version for user to select.

## Pen Input Handling Services

Touch screen panel inputs are another form of users' input method. Pen input handling services provide APIs to create "active area" and means to handle touch screen inputs.

## Handwriting Recognition Input Handling Services

Handwriting recognition input handling services handle a special type of pen inputs. Inputs collected by HWR services are passed to a HWR engine for recognition and the results are handle back to the system for further processing..

## Application Framework Services

PPSM-GT supports application framework such as Metrowerks PowerParts. PowerParts is an optional application software available for users that would like to develop their application with a application framework.

PowerParts is an object-oriented framework designed for embedded application programs. It helps to create and customize an application’s graphical user interface (GUI), including both behavior and appearance.

Because PowerParts is an object-oriented tool, UI developers can inherit and customize the UI design’s behaviors and appearances from PPSM-GT’s default UI’s component library. This would reduce the UI design cycle time. Customization of components’ behavior and appearance can be done when needed.

PowerParts also comes with an X86 component library that allows developers to design and develop their UI in the PC environment. This means that developers could develop and test their UIs in the PC environment even if the embedded hardware is not available.

PowerParts provides a skeleton for developing graphical user interfaces and includes the following:

- Application event handling
- View hierarchy
- Default controls and appearances
- Graphics engine
- Resource management

For more information on PowerParts, please refer to PowerParts User Guide.

## In A Nut Shell

PPSM-GT provides:

- A multitasking, real-time executive for the Motorola MC68VZ328 (DragonBall VZ) and other microprocessors.
- A full-featured, compact ROM operating system.

- A priority-based, pre-emptive and post-emptive kernel for task scheduling and time-slicing multitasking support.
- Services for handling kernel activities, task manipulation, memory and power management, events, software timer functions, audio, RTC, alarm, UART, and touch screen panel support.
- User application download support.
- Message and event broadcasting.
- Dynamic run-time task creation and deletion.
- IrDA communication using OBEX and IrCOMM.
- A TCP/IP stack for e-mail and World Wide Web activities.
- Graphic and User Interface services for handling LCD display and user inputs.

PPSM-GT is a complete software product for creating embedded real-time applications. It is a system that has a multitasking kernel and many field-proven routines that are designed for PDA and portable device developers. PPSM-GT will make development work easier and more efficient, helping to increase competitiveness in the fast-moving twenty-first-century market.

# Section 1

# Background

This section covers concepts that are helpful for writing PPSM-GT code. It is intended to provide a basis for understanding PPSM-GT. Detailed information on PPSM-GT is presented in later sections of this user guide.

Before designing application with PPSM-GT, developers need to decide how they like to use PPSM-GT. Like any other software tool, proper use of the tools will result in much benefits and productivity. The strength of PPSM-GT is that it could either be used by itself or combine with a application framework to develop applications. It is important to decide which method to pursue as the approaches are different.

If the approach is to develop application using application framework, then the developer should approach their design as specified in the framework and used PPSM-GT services as supporting the framework applications. Developers then need to focus on the framework application manual when designing the user interface and refer to PPSM-GT user guide when designing the low level routines (sometime known as “engine”) for the applications.

Another approach developers could adapt is to use only the PPSM-GT and no other application framework to develop applications, Chapter 3, “PPSM-GT Core Programming Concepts” provides the basic PPSM-GT concepts that are uniquely defined, and also explained the 3 approaches to design with PPSM-GT.

This section consists the following chapters:

- Chapter 2, “PPSM-GT Core Fundamentals”—introduces fundamental, required PPSM-GT concepts.

- Chapter 3, “PPSM-GT Core Programming Concepts”—introduces fundamental, required PPSM-GT core programming concepts, and describes the 3 approaches for designing with PPSM-GT.

# 2 PPSM-GT Core Fundamentals

This chapter provides a fundamental overview of the PPSM-GT's core. It is intended for those who are unfamiliar with the PPSM-GT kernel and its system services, which are the basic building blocks of PPSM-GT's core.

This chapter is divided into the following sections:

- Kernel Fundamentals—describes fundamental concepts about kernels.
- System Fundamentals—describes fundamental concepts used in the PPSM-GT system, such as application environments and the system environment.

## Kernel Fundamentals

A real-time kernel (sometimes known as a real-time executive) is software that manages system resources. It also provides an architectural framework for application software that is based upon a defined set of design policies and operational rules. The kernel is a software component and is not considered a part of the application. At a minimum, it contains a scheduler (used to determine what application program is to gain control of the CPU) and a library of services that are invoked by the application programs. Each service in the library operates on one or more data structures, which are called objects, to achieve desired operational behavior. The *real-time* prefix is added when the kernel is designed and implemented in such a way that it is suitable for use in applications that are time-critical.

Tasks are a series of routines that can execute concurrently to implement the application design. Multitasking promotes optimal use of the CPU while providing seemingly concurrent task execution.

Setting priority levels allows the user to define the relative importance of tasks. Event-driven operation allows the kernel to respond predictably to events as they occur.

These are some of the basic concepts that are covered in this section. The objective is for users to achieve an understanding of these terms and concepts, which are the building blocks of kernel services. The following are the section's topics:

- Tasks
- Multitasking
- Priority
- Pre-Emptive and Post-Emptive Task Switching
- Time-Slicing Operations
- Event-Driven Operation

## Tasks

In a real-time embedded system, the system integrator or software developer divides the overall function of the application into smaller entities called tasks. A task is a basic working unit of user code that performs a defined function or set of functions in the application design. Each task operates independently of other tasks but can establish relationships with other tasks. These relationships can exist in the form of data structures, input, output, or other constructs.

A task executes when the real-time kernel determines that the resources required by the task are available and that no other task of higher priority is also ready to run. While running, a task controls all system resources. However, because the system can have many tasks, a single task cannot be allowed to control all of the system resources all of the time. The need to service other tasks requires multitasking.

## Multitasking

Multitasking promotes optimal use of the CPU and gives the appearance that a single processor can perform multiple operations concurrently. In reality, it is impossible for a device that executes instructions sequentially to perform two or more operations at once. However, if system functions are separated into different tasks (and given the relative speed difference between the physical process and the CPU), the system switches quickly from task to task, and the effect of concurrency can be achieved.

In multitasking, each task executes until one of four possible things happens:

- The task completes its function and self-terminates.
- The task is suspended while waiting for an event to occur.
- The task self-suspends as a resource that is needed is unavailable.
- The task is interrupted by other higher-priority tasks.

Because the CPU is a valuable resource, it is used most efficiently when it is performing productive operations and is not idle. If a task that is in control of the CPU is no longer able to use the CPU productively, the kernel will switch control of the CPU to a ready task that can use it. When done at a rapid rate, this switching between tasks gives the appearance of several tasks being executed simultaneously.

## Priority

Priority defines the relative importance of each task. A multitasking real-time kernel maintains an orderly transfer of CPU control from one task to another by keeping track of the resources needed by each task, as well as its priority and state, so that execution of the task occurs in a timely manner.

During system operation, application tasks compete for system resources such as memory, the CPU, and peripheral devices. A task should not monopolize a system resource if a task of higher priority requires the same resource. To prevent this occurrence, and to achieve execution timeliness, each task has a priority that the kernel uses to determine a task’s place within the sequence of other runnable tasks.

Tasks of low priority can have their execution pre-empted by a task of higher priority.

## Pre-Emptive and Post-Emptive Task Switching

A pre-empt occurs when a currently running task must give up its execution rights to a higher-priority task. The running task is said to be pre-empted by a higher-priority task. The pre-empted task is blocked by having its registers saved in the task stack region, but the task remains ready to run.

A post-empt occurs when the running task executes a directive to block the task itself. (*Post-empt* is a coined word that implies the converse of a pre-empt.) A post-empt could occur when a task performs, for instance, the Event Get API when no events are in its queue. The task is then blocked and a post-empt occurs. Similarly, any self-imposed “suspend” or “wait” API such as KnlSuspend or KnlWait() may cause a task to be post-empted.

PPSM-GT supports both pre-emptive and post-emptive task switching to allow valuable CPU resources to be passed to higher-priority tasks when required.

## Time-Slicing Operations

A time-slicing operation is a special kind of priority task operation. A time-slicing operation occurs when there are two or more tasks that have the same priority level; these tasks are called time-sliced tasks. During a time-slicing operation, each task is given a fixed time duration, or quantum, for execution.

The quantum time interval for time slicing is set in the configuration file and fixed at runtime.

### Normal Quantum Operation

In PPSM-GT, time-sliced tasks are grouped together, and the sequence of their execution is based on the time of creation. For example, if task A and task B have the same priority and task A is created before task B, then task A will execute first, followed by task B.

Assuming tasks A and B are time-sliced tasks and task A is currently running, if the quantum for task A has expired, task A will be blocked, and the next time-sliced task, task B, will be started. When the quantum is available for task A to run, task A will continue to run from the point at which it was stopped; it will not restart at the beginning.

*Given quantum* is defined as the predefined time period allocated for the time-sliced tasks to operate. In normal operation, a time-sliced task's execution is restricted to be within the given quantum. However, under certain conditions, time-sliced execution could execute in less than or more than the time of the given quantum. These conditions are highlighted in the next two sections.

### Time-Sliced Task Operating in More Time Than Given Quantum

There are two cases in which a time-sliced task might get more operation time than the given quantum.

**Case 1:**

When a time-sliced task is pre-empted by a higher-priority real-time task, this time-sliced task will be run when all higher-priority real-time tasks are blocked again, and it will execute with a new quantum. The total time for the given time-sliced task will be 2Q – X where Q is the quantum time and X is the time taken by the higher-priority tasks.

**Case 2:**

If a time-sliced task yields its quantum, the next time-sliced task will be given the remainder of the running task's time slice in addition to its own quantum. No time-sliced task can have more than two continuous quantum periods. The total time for the time-sliced task is 2Q – Y where Q is the quantum time and Y is the time yielded by the yielding task.

### Time-Sliced Task Operating in Less Time Than Given Quantum

Under the following conditions, a time-sliced task will operate in less than the full given quantum:

- A time-sliced task is self-terminated before the given quantum.
- A time-sliced task yields it quantum.
- A time-sliced task operates in less than its quantum because another task self-terminates.

Other conditions under which a time-sliced task will operate in less than the full given quantum can involve multiple tasks. For example, assume that there are three time-sliced tasks—task A, task B, and task C—that are all ready to run. Task A is first to execute, followed by task B and then task C.

- If, during task A execution, the task is blocked by itself, task B will start executing in the remainder of the task A's quantum only.
- When the quantum expires, task B will stop its execution, and task C will start execution with a new slice period or quantum.

The total execution time for task B is only Y, where Y is the time yielded by the yielding task.

## Dynamic Time-Sliced Task Creation

In PPSM-GT, time-sliced task operation can be created or terminated statically and dynamically.

**WARNING!** Time-critical priority tasks could be changed to time-sliced tasks at runtime.

Because PPSM-GT allows task creation and priority levels to be changed dynamically, a task might be created at compiler time as a non-time-sliced task, but become a time-sliced task during runtime because another task was created dynamically with the same priority.

Therefore, it is good programming practice to track and limit the priority of tasks that are created dynamically in order not to create unnecessary runtime errors, such as the conversion of time-critical time-prioritized tasks into time-sliced tasks.

## Event-Driven Operation

Event-driven operation allows the kernel to respond predictably to events as they occur. An event can be any stimulus that requires a reaction from the kernel or a task. Examples of events include timer interrupts, alarm conditions, and keyboard input.

Events may originate externally to the processor or internally from within the software.

PPSM-GT provides two groups of events for task interworking: normal and broadcast events.

Normal events are sent on a one-to-one basis, whereas broadcast events are sent to tasks that are in the same broadcast channel. A task therefore has to be connected to the broadcast channel in order to receive broadcast events.

Channels are the media on which the information is being carried. Only the tasks on the same channel can receive the event being broadcasted. A task needs to connect to the channel to receive events.

For broadcast events, the event pointer is relayed from one task to the other that belongs in the same channel. It begins with the highest priority task in the channel and relays down to other tasks in the channel based on priority.

# System Fundamentals

PPSM-GT system fundamentals are essential elements that are responsible for setting up an operating environment for the PPSM-GT system. A good understanding of basic concepts such as applications, graphic contexts, input contexts, and tasks enhances developers’ mastery of writing applications with PPSM-GT system layer services.

The following topics are discussed in this section.

- System Applications
- Graphic context, GC
- Panning Screen

- Input Context, IC
- Active area
- Pen Input Area
- LCD Display Screen
- Hardware Cursor
- Relationship of Application, Task, Panning Screen, Graphic Context, Input Context and Active Area
- Some common examples and pitfalls

## System Applications

Application and function are two word that are commonly used interchangeably especially when referring to a product. For example, "How many applications does the PDA have ?" or "How many functions in the PDA?" normally generates the same answer.

In PPSM-GT, it is useful not to mix the two words: application and function together. Application defines as the characteristics of the product whereas function defines the behaviors or operations of the product.

Take the example of a simple PDA product, it is a PDA because it has applications such as calculator, scheduler, alarm and real time clock, and in order for it to behave like a PDA, it needs functions that could preform calculating, scheduling, alarm and real time clock activities.

In PPSM-GT, applications are like stages in a drama where all the actions are taking place, whereas tasks are like casts who preform all the functions in the drama.

## Graphic context, GC

The graphic contexts are memory buffer that are used to store the property such as style, color, size etc. for the task. The relationship of GC, task and application are important for contents to appear on the LCD display. In PPSM-GT graphic context GC is one of the building block for displaying information.

## 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 and bound to the application. Pen Input areas on the panning screen will receive pen input data only when they overlap with the LCD display screen.

## Input Context, IC

Input contexts are the input property for the touch screen panel. They are also memory buffer that stored pen input time-out, sampling rate, pen size, pen color, software keyboard, active area list, and task that is bound to the input context. The relationship of IC, task and application is important for system to receive any input from input pad. In PPSM-GT input context, IC is one of the building block for receiving information.

## Active area

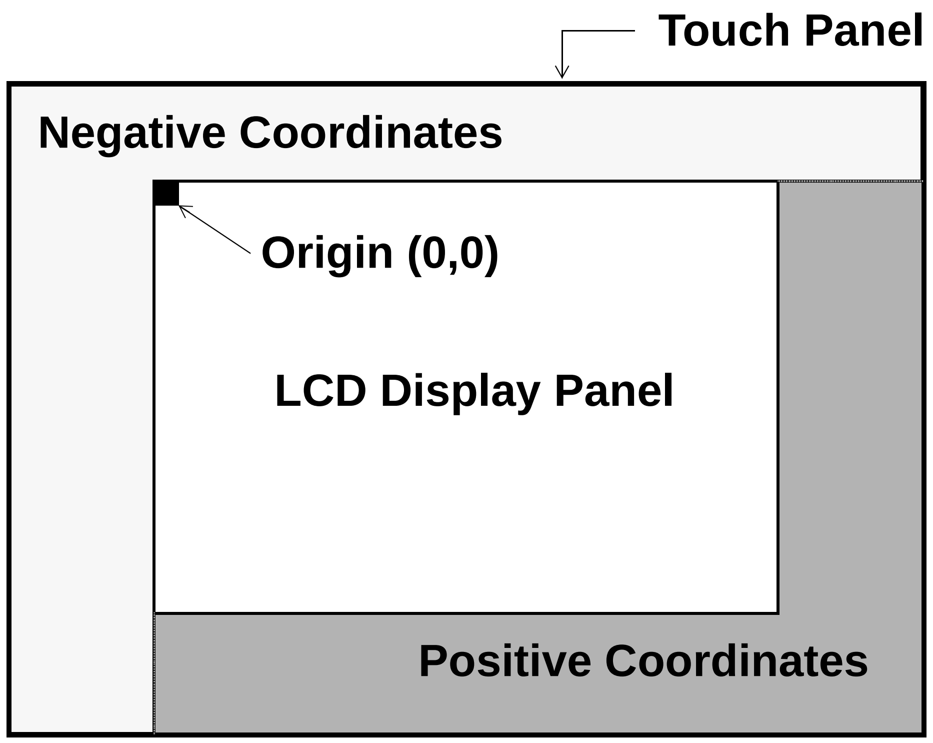
An active area is defined as a rectangular region of the pen input area where an application or an action will execute if the region is pressed. An example of this is an icon, or an action button.

Active areas are classified into two groups, icon area and input area.

## 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 2.1, the input coordinates outside of the LCD display screen can have a negative value. PPSM-GT allows negative coordinates for pen input. This allows applications to implement features such as off screen icon and off screen writing area.

**Figure 2.1 PPSM-GT 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.

## LCD Display Screen

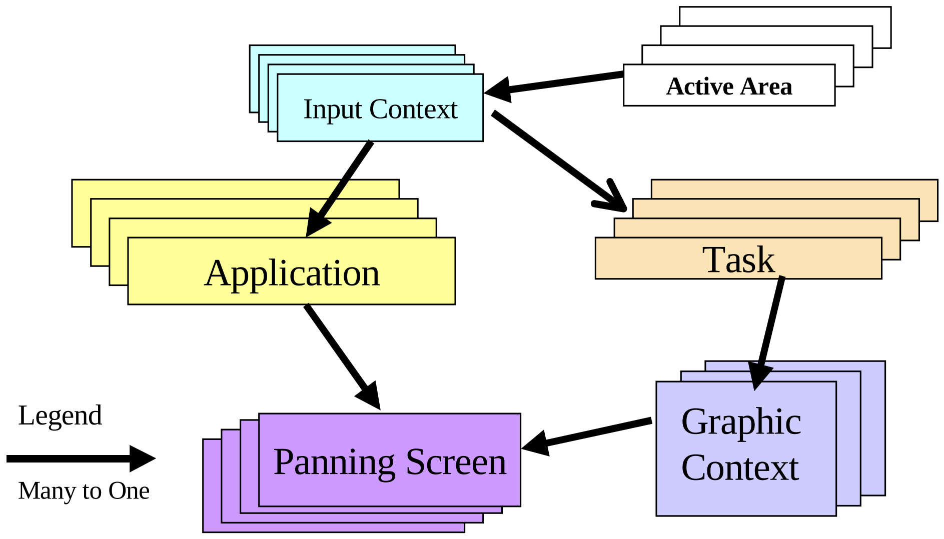
The display screen is the LCD display area where applications can display images. The LCD module can handle both 1, 2 and 4 bits per pixel graphics, giving black and white 2 display, 4 or 16 grey levels display respectively. Displayed data, such as graphics and text, can only be seen within the display screen area.

## 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's cursor status, position, size and the offset of display origin on panning screen will be used.

## Relationship of Application, Task, Panning Screen, Graphic Context, Input Context and Active Area

**Figure 2.2 Relationship of Application, Task, Pan Screen, GC, IC & Active Area**



Panning screen, active area, graphic context, input context, task and application are all independent components in PPSM-GT such that each component need to be created separately. They are, however all related together for system operation. A typical program written with PPSM-GT will consist of many GCs, many ICs, many tasks and several applications.

Figure 2.2 shows the relationship of application, task, panning screen, graphic context, input context and active area. The arrows highlight the many to one relationship. For example when the arrow points from application to panning screen, that means that many application could be using the same panning screen. In most cases the reserved is not true, that is many panning screen cannot be linked to one application all at the same time. The following sections will highlight the main features and restrictions.

In short beside the requirements and restrictions that application needs panning for display and task needs IC for input, and GC for drawing, and application needs tasks for action, there is no hard and fast rules on the usage of applications, tasks, graphic context and input context. PPSM-GT has provide much flexibility and open up area of design to the user's creativity. The key is to be exercise control and be creative.

### Application and panning screen

Panning Screen are memory buffers for display purpose. For example, to display an image of the ship, the image of the ship is stored in the panning screen and displayed onto the LCD screen when the panning screen becomes active. By default each application is bound with a default a panning screen, and when the application is swapped in by the system, the default panning become active. An active panning screen displayed it's content onto the LCD display as it is written.

The relationship between application and the panning screen is many to one; i.e. Many applications could be pointing at the same panning screen. The advantage of such arrangement is for memory resource saving when there is no need to have new displays when switching from one application to another. The disadvantage of such arrangement is that the graphic are overwritten each time a new graphic is drawn. If there are no provision to save the old graphic, then it is lost.

Calling AppSwitch() API will cause the change in panning screen if the applications have different panning screen.

### Application, task, input context (IC), and active area

Input contexts are memory buffer that stored pen input time-out, sampling rate, pen size, pen color, software keyboard if exist, active area list, and task that is bound to the input context.

Input context are tied to task as those inputs captured by the system will be sent to the task specified in the input context. The relationship is many to one; i.e many IC could be having the same task Id and each IC can only have one task Id.

Active area are area on the touch screen panel that takes movements when touched. The relationship for application, task, input context, and active area is illustrated with the following situation.

When there are inputs at the touch panel, the system alerts the application through an event that there are active area inputs. The application then search through it's input context list to determine which active area is touched. Then  an event will be sent to the task which task id is in the input context whose active area has been touched.

### Task and graphic context

In PPSM-GT tasks need graphic context for drawing. PPSM-GT allows one or many tasks to be bounded to one or many GCs. That is each task could have individual GC or tasks could share GC provided each task is limited to one GC at any one time. GC however is not necessary tied to only one task. At any one time, one GC could only be bound and used by one task. Tasks are allowed to share graphics context.

Also not every task will have a graphic context. It is based on a need to have basic as not every task is required to draw. If a task do not have to draw or do not have draw any more, it should not have a GC.

### Task and Input context

The relationship between task and IC on the other hand is different. Each task can have many IC and, input context cannot be share among task. That is each IC can only be bound to one task at any one time. Tasks are however, allow to unbind those input context that are not used and other task are allowed to bind unused IC.

In an application there are input contexts which are bound to tasks. In this case, whatever event occurred in the active area inside the input context, event will be generated and the system will then determine which task this event will be sent to.

## Some common examples and pitfalls

The following are some of the common examples in using applications, graphic context and input context. It also highlights certain pitfalls that are commonly encountered. Please take note to avoid repeating the same problem.

### Example 1: Tasks with different graphic context but same panning screen

Tasks may require to perform several functions as such it will have several graphic contexts for different drawing properties like dot width, fill pattern, etc. However, if the GC has the same panning screen, then the drawing will be put in the same memory area allocated for the panning screen, and if the panning screen is bound to the current application, then the content in panning screen is displayed on LCD. Whatever that is drawn by the tasks will be seen on the display.

### Pitfall 1: Wrong display on LCD

Graphic contexts that use the same panning screen will have whatever drawn by the tasks be seen on LCD if the panning screen is bound to the current application.

### Example 2: Applications with same panning screen

Applications are allowed to share panning screen, therefore in some cases one panning screen may be share by two or more applications.

### Pitfall 2: Same display on application switching.

In application switching, the LCD will remain the same if the panning screen is the same. There may be cases that the same image remains when the program switch from one application to another. For such case, when same panning screen is used for different application care must be taken to clear the display and redraw new graphics.

## Summary

This chapter provides an overview of the fundamental concepts of kernels, system layer services and framework services.

Kernel fundamental concepts include tasks, multitasking, task switching, and event-driven operation.

System layer fundamental concepts include applications, graphic contexts, input contexts, active area and panning screen.

3

# PPSM-GT Core Programming Concepts

This chapter builds on Chapter 2, PPSM-GT Core Fundamentals. It discusses the programming concept of PPSM-GT's core.

The following topics are discussed in this chapter:

- The PPSM-GT Programing Approach
- The Typical PPSM-GT Application Program

## The PPSM-GT Programing Approach

PPSM-GT applications are generally multitasking, event-driven programs. One or more tasks could be running at the same time. To successfully build a PPSM-GT application, understanding on how the system itself is structured and how to structure the application are essential.

In some simple systems, one application may be sufficient to launch all features of the product, whereas in other systems, a multiple-application environment is required. There is no absolute rule on the relationship between the number of applications and the features of the product. PPSM-GT allows different approaches to designing systems. For more information on system design, refer to Chapter 11, System Application Services.

When developing applications using the PPSM-GT, the developers must first decide on how they are going to use the PPSM-GT services before designing and writing the main program. There are many approaches to design applications using the PPSM-GT, the commonly used 3 methods are:

1. Single Platform Approach
2. Dual Platform Approach
3. Multi Platform Approach

### Single Platform Approach

The single platform approach assumes that developers are using only PPSM-GT standard services to design all the applications including user interface. In this approach the graphics routines are used to design the appearances of the applications and the user interfaces are design from the standard libraries. No application framework is used and C is developing language.

Under this environment, the main program is much simple. The main function of the main program would to create an environment for the tasks to execute. Normally the main program would create all the system applications required, and a few key tasks that control the smooth execution of the whole system.

**NOTE** Most of the chapters in this user guide is written assuming that the single platform approach is adapted. In cases where dual or multi platform are used it will be specified.

### Dual Platform Approach

The dual platform approach assumes that developers are using another application framework to develop the user interface and use the PPSM-GT system services to develop the rest of the application. In this approach, no or very minimium PPSM-GT default graphic routines are used. All graphic are handled by the framework and the framework is an application layer sitting on top of the PPSM-GT. In such situation, UI design are done in the framework environment and low level routines(sometimes referred to as engine) are design using the PPSM-GT. There could aso be possible that UI is developed in another language such as embedded Java or C++ .

Interworking between the two environment becomes critual in such design. PPSM-GT has suggested some basic requirement in the next section, The Application Framework Requirements. These are provide as guidelines and could varies with application framework. Developers have to check the what is required by the application framework and design according to the requirements.

### The Application Framework Requirements

If an application framework is used to develop the user interface then to support the application framework in the PPSM-GT environment, the following may be one of the way the PPSM-GT interwork with application framework. The following tasks are added in the PPSM-GT main program:

- The Framework Task
- The System Event Task
- The idle Thread

Figure 3.1 shows the steps in PPSM-GT programming for designing the main program for a system using a application framework. In this example, there are two basic tasks that need to be created before the system can use the application framework. These two tasks are one of the ways the application framework could employed to interwork with PPSM-GT. They are not necessary if the application does not use any application framework.

**NOTE** .The next 3 sections are related to designing application using application framework in PPSM-GT environment. Skip these sections if an application framework is not used.

### The Framework Task

The Framework task is a special task that needs to be created for the framework application. This is a system task to register the framework to the system. The following are the functions performed in the framework task:

1. Set up system display properties such as input and output media.
2. Calibrate input area if required.
3. Mark the area to be the input for the touch screen panel.

4. Set up the framework X-Y display dimensions according to the LCD screen and the pixels supported.
5. Set up the default system font.
6. Create the event dispatcher and event loop.
7. Create the display properties such as color, color palette, and graphics port.
8. Create a location-tracking object such as a mouse object.
9. The framework has been set up for drawing graphics.
10. Resume the suspended system event task.
11. Start the Framework idle thread if required.
12. Wait Loop.
13. Clean up the Framework parameters when exiting the Framework task.

### The System Event Task

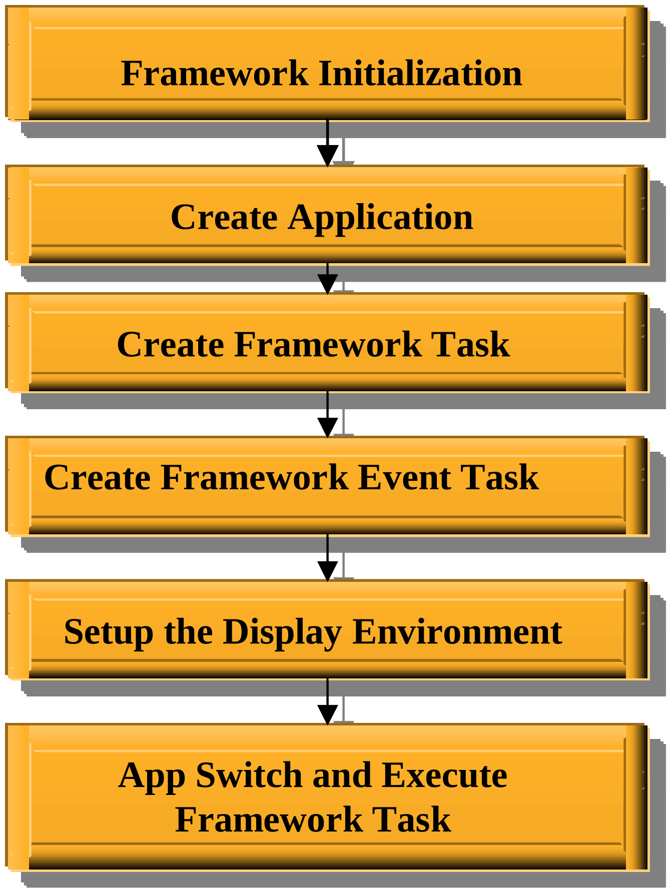
The system event task is another special system task created for the framework to communicate with other PPSM-GT core system tasks (and vice versa).

This system event task handles all system events that are designated for the objects of framework. It checks for the pen input events that the Framework application is expecting.

### The idle Thread

In some application framework, a idle thread is required. The idle thread is the framework application's wait loop. A framework application waits in the idle thread when there is nothing to do. Each PPSM-GT application that uses the application framework that requires a idle thread must have the idle thread. That is, if, in the multiple-application environment, there are 5 system applications that use the application framework, then there will be 5 Framework tasks and System Event tasks running, with each framework having it's own idle thread.

**Figure 3.1 Basic Steps for PPSM-GT Main Program with Application Framework**


Framework Initialization
Create Application
Create Framework Task
Create Framework Event Task
Setup the Display Environment
App Switch and Execute Framework Task

## Multi Platform Approach

The multi platform approach is the last of the 3 platform approach and is the most complex and if developer choose to use this platform approach, much care has to be exercide. This approach is a combination of the 1st and 2nd approaches. That is application framework and default PPSM-GT graphic routines are both used to handle the application graphics.

Only advance developers who are experiences with PPSM-GT programming are encourage to use this approach. Care must be taken to ensure that when switching from the framework environment to the PPSM-GT environment and vice versa, all resouces especially those related to graphic rouines are accounted for. Misappropriate handling will result in undesirable behaviour of the system.

## The PPSM-GT Main Program

The PPSM-GT main program functions like the main in C program. At the bare minimum level, the system needs to have a main program. Upon power up, after performing the system initialization routines, the system calls the main function to launch the application. Therefore, in the main function, at least one task must be created to launch the system application. The number of system applications to be launched in the main program depends on the complexity of the system.

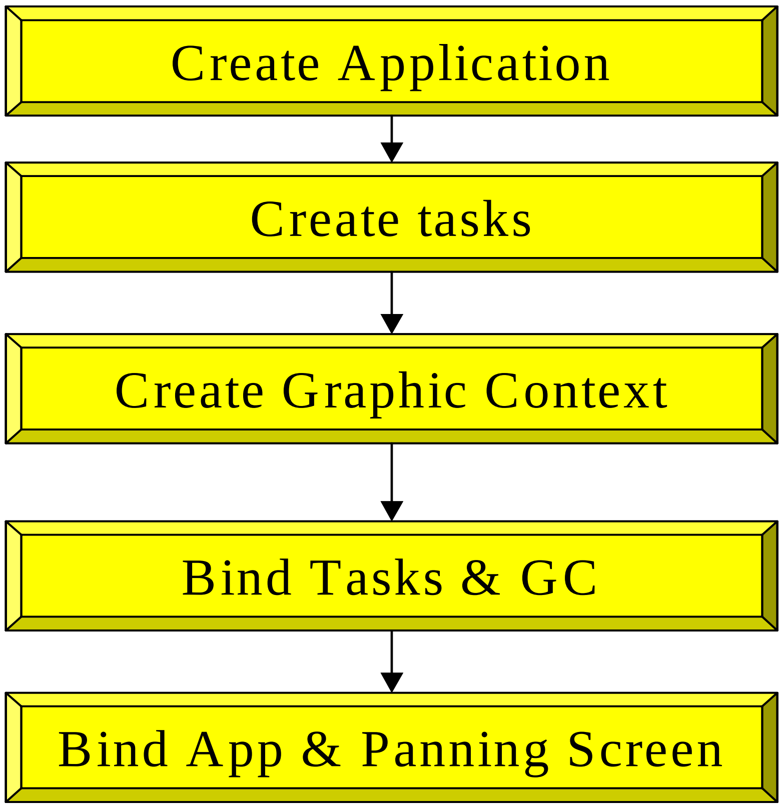


**Figure 3.2 Main Program flow of a single platform approach system**

The flow chart in Figure 3.2 shows the steps for the main program flow for single platform approach. Note that instead of the framework task and the system event task, the graphic context and panning screen are created. The rest of the process remain the same. The number of steps varies with the number of applications and tasks in the system. For example, if in the system there are 10 applications and 50 tasks, then AppCreate() will be used 10 times and KnlCreateTask() will be used 50 times.

Each application and task on the system must be registered with PPSM-GT before the application or task can make use of the PPSM-GT tools. Using AppCreate() and KnlTaskCreate() will register the application and task to the system. Registration ensures that the runtime memory and stack required for each application are allocated within PPSM-GT's memory system.

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

After all applications have been registered, the first application can be started by calling the AppSwitch() API. The system will exit the main program and start executing code in the designated application specified by the AppSwitch() API.

## The Typical PPSM-GT Application Program

This section describes the general flow for a typical system application program in the PPSM-GT environment. Not all applications will have the typical flow shown in Figure 3.3. In some applications, one or two steps may be added or deleted. It all depends on the requirements of the application.

After the main program, the system has registered all system applications and starts to execute the application code specified in the corresponding application. In a typical application, there will be an event checking loop that checks for incoming events when there are no other activities to perform. When an event occurs, the task processes the event and might involve application swapping and so forth. In most cases, the task waits in the event checking loop.

## Event Checking Loop

In general, all event driven tasks in PPSM-GT should have an infinite wait loop for when the task has completed necessary functions and is waiting for an event. Event checking loops are defined as system waiting points where each individual task communicates with the kernel.

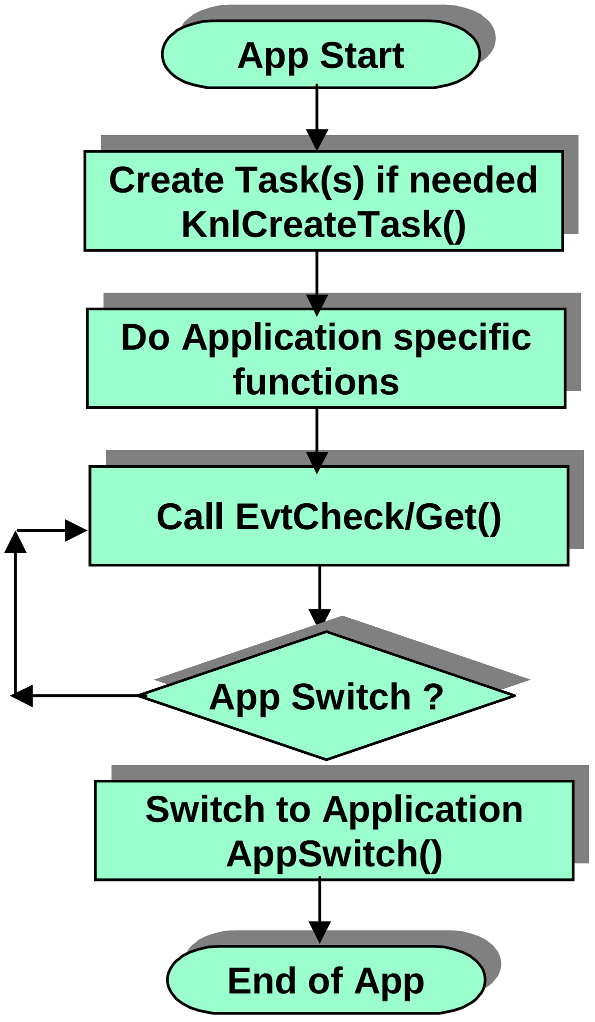
PPSM-GT provides two APIs, EvtCheck() and EvtGet(), to get new events on the event queue. (Note that the PowerParts application uses different commands to wait for events.) EvtCheck() tries to get the new event from the event queue and not block the task execution, whereas EvtGet() blocks the task execution and waits for the new event.

In most cases, the event checking loop consists of a set of case statements checking against the event header. (For details on event headers, please refer to Chapter 12, System Event Management Services.) Based on the event header, the task decides on the next actions.

Properly set up tasks should not have local loops where program loops are indeterminately waiting for some external action. Such cases do not allow the kernel to schedule other tasks, and they waste processor time. An exception might be a very small delay for hardware settling time.

In cases when task self-termination is required, having a return command inside the task routine will cause the current task to self-terminate. Figure 3.3 shows the application flow.

**Figure 3.3 Typical PPSM-GT System Application Flow Chart**



## Event Messages from PPSM-GT

PPSM-GT applications should take a pro-active role—that is, an event-driven approach. When external events occur, such as the pressing of the input panel, the PPSM-GT system automatically intercepts and interprets the external event. If the event requires attention from the application, such as when an active area is pressed or there is incoming data from the UART, PPSM-GT sends an event message to the waiting task for processing.

However, if the external event is not intended for the application—as in the pressing of an input panel that is not defined as an active area, or when there is a time-out for going into power-saving mode—the event is handled by PPSM-GT internally. PPSM-GT may send a non-wakeup event message to the tasks, or it may send nothing to the tasks, depending on the situation. Refer to Chapter 12, System Event Management Services for more information on system events.

## Code Sample

This example gives an idea of how to create tasks, register tasks, create an application, and register an application in the PPSM-GT environment.

Listing 3.1 shows a main program for single platform approach running on a PPSM-GT operating system on a Motorola MC68VZ328 (DragonBall VZ) embedded microprocessor.

**Listing 3.1 Main Program Example**

```
/* Creates an application ABC with a pannning
screen and two tasks; menu task and Draw task  will
have a graphic context  having the same panning
screen of the applications */
#include "OS.h"
#define MenuTask 0
#define DrawTask 1
#define MENU_PRIORITY 6
#define TASK_PRIORITY 4
#define DEFAULT_MODE 1

U32     gpAppId;
U32     gpTaskId[2];
U32     gpGCId;
U16     gPanScnWt, gPanScnHt;

STATUS main(void)
{
const TEXT AppName = {'A', 'B', 'C', 0};
const TEXTMenuTaskName= {’M’,’E’,’N’,’U’,0};
const TEXTDrawTaskName= {’D’,’R’,’A’,’W’,0};


U32 TaskAddr[2] = { MenuTask, DrawTask};
S8Priority[2]={MENU_PRIORITY,TASK_PRIORITY};
volatile U8          i;
volatile STATUS      status = SYS_OK;
volatile SCREEN_ID   screenId;
volatile EVTPORT_ID  portId;
GC_ID                gcId;


/* Creating an application ABC with no
constructor, destructor, entryCallback and
exitCallback*/

status = AppCreate(&gpAppId, AppName, 0, 0, 0, 0,
0,3000);

/* creates graphic context with a panning screen
*/
    GpxCreateGC(&gpGCId, 160, 240);

/* Creating 2 task MENU with priority of level 6,
DRAW task at priority level of 4 and default mode
of 1 and stacksize of 10240 bytes*/

    status = KnlCreateTask( &gpTaskId[0],
(P_VOID) TaskAddr[0], MenuTaskName, 1024,
Priority[0], DEFAULT_MODE);

    status = KnlCreateTask( &gpTaskId[1],
(P_VOID) TaskAddr[1], DrawTaskName, 1024,
Priority[1], DEFAULT_MODE);

/* Suspend the task MENU for now*/
    status = KnlSuspend(gpTaskId[i]);

/* Set the pen ring buffer for ABC app to 2000
bytes   */
    PenSetRingBuffer(gpAppId, 2000);
```

```
/* Bind Gc with task MENU */
    status = KnlBindGC(gpTaskId[i], gpGCId);

 /* Bind the same panning screen to the
application */
    status = GpxGetPanScreen(gpGCId, &screenId);
    status = AppBindPanInfo(gpAppId, screenId);

    }

/* Switch to the ABC application to enable the
graphic to appear on the LCD and resume the
suspended task MENU*/

        AppSwitch(gpAppId);
        KnlResume(gpTaskId[0]);

/* Report any error while creatung application,
task or GC */

        if (status != SYS_OK)
          return SYS_ERR;

        return;
}
```

## Summary

This chapter gives programmers information on how to start programming in PPSM-GT. The chapter discusses application and task registration as well as event loop checking. Figure 3.2 gives an idea of the actual program flow in PPSM-GT.

# Section 2

# Getting Started

The chapters in this section cover tasks that must be performed before a programmer can start designing the system with PPSM-GT. The following chapters are contained in this section:

- Chapter 4, "Installing PPSM-GT"—provides instructions for installing PPSM-GT, and defines the hardware and software requirements for both the development and runtime environments.
- Chapter 5, "Developing a New Application"—provides information with which all PPSM-GT programmers must be familiar before writing any code using PPSM-GT, and describes how to set up a new CodeWarrior project for a PPSM-GT application.
- Chapter 7, "ISR Routines Services,"—provides informations on how PPSM-GT handles interrupt software request. It is used as a device driver interface between the application and the hardware. It provides information on the making of drivers that are built on the top of DragonBall's interrupt controller configurable

4

# Installing PPSM-GT

Before starting software development with PPSM-GT, the programmer must have the necessary hardware and software. This chapter covers the following topics:

- PPSM-GT Requirements—system requirements for application development and for running a completed PPSM-GT application.
- Installing PPSM-GT—refers to resources explaining how to install PPSM-GT and which directories and files are installed.
- PPSM-GT Documentation—refers to PPSM-GT documentation resources.

## PPSM-GT Requirements

There are two kinds of requirements to consider: software and hardware.

### Software development requirements

PPSM-GT development requires a PC running Windows 95, Windows 98, Windows NT Workstation 4.0, or Windows 2000. The memory and hardware requirements to develop PPSM-GT applications are the same as those for running the CodeWarrior IDE on a Windows PC. More RAM may be required if the IDE and an application are run concurrently.

**NOTE** Refer to the *CodeWarrior IDE User Guide* for more information on the requirements for running the CodeWarrior IDE.

### Hardware development requirements

The MC68VZ328 Application Development System (ADS) is used as the reference development platform throughout this user guide. The A/D convertor used is a 10-bit component, giving a resolution of 1024 in X and 1024 in Y.

For details on the hardware configuration, please refer to the *MC68VZ328 ADS User's Manual.*

## Installing PPSM-GT

PPSM_GT installation procedures are described in the *Release Guide.*

Information about PPSM-GT contents and distribution is contained in the "readme.txt" file in the `root\PPSM-GT` directory. The *Release Guide* also contains information about the recommended hardware for development and application examples.

The installer program automatically copies all PPSM-GT files necessary for development to the `root\PPSM-GT` directory on the development machine's disk. PPSM-GT release notes for a complete description of the PPSM-GT directory layout.

## PPSM-GT Documentation

The PPSM-GT documentation is located in the `PPSM-GT\documentation` folder on the PPSM-GT CD. The PPSM-GT documentation consists of the *PPSM-GT User Guide* and the *PPSM-GT API Reference*. Each of these documents is in `.PDF` format.

## Summary

PPSM-GT development requires a PC running Windows 95, Windows 98, Windows NT Workstation 4.0, or Windows 2000.

A finished PPSM-GT application requires a device that provides a video device for output and at least one input device, such as a mouse or pen.

The installer program installs all files needed to build a PPSM-GT program onto the development PC.

5

# Developing a New Application

PPSM-GT can use both Metrowerks CodeWarrior Integrated Development Environment and Single Step SDS as it's development environment. The way to develop a new application does not depend on the type of development tool used, but the setup of the environment does. Both tools have different approaches to the development environment, the SDS used the command based method whereas the CodeWarrior IDE provides editing, linking, debugging, and complying all in one environment. The materials covered in this chapter provides only a overview of the development tool environment. It is not intended to teach the usage of the development tool. Please refer to the individual development tool manual if more information is desired.

This chapter contains only an overview of the SDS Singlestep and Metrowerks IDE development environment, and the PPSM-GT application development which are as followed:

## SDS Singlestep Development Environment

One of the focus point for the SDS singlestep development environment is the makefile, which is a script file that consists of command and options for the various development tools such as complier, linker and etc. A make program is required to execute the makefile. PPSM-GT and SDS singlestep do not provide the make program, the developer have to get their own make program.

A sample make file is as shown in Listing 5.2 , To ensure that the development environment is setup correctly, all paths and options in the makefile must be defined correctly. The make program will execute the SDS Singlestep tools based on the commands and options in the makefile to carry out the instruction.

In the PPSM-GT, there are 3 types of makefile and they are:

- PPSM makefile -- use only if the source files of the PPSM-GT libraries are modified.
- Device driver makefile - uses only if modification of device drivers are made.
- Application makefile - most commonly used and for build user application program.

All the makefiles are stored in each individual directory. The most commonly and frequently used makefile is the application makefile.

Another focus point of the SDS Singlestep environment is the debugger which provide the debugging environment for the software to be tested and evaluated. Please refer for the SDS user guide for more details on the tool.

## Metrowerks IDE Development Environment

The focus point of the Metrowerks IDE is that it permits a software developer to quickly assemble source code files, resource files, library files, other files, and configuration settings into a "project," without writing a complicated build script (or "makefile").

Source code files may be added or deleted from a project using simple mouse and keyboard operations instead of tediously editing a build script.

This section intends to use the configuration of a single project to teach the developers the method to create and manage several configurations of settings for use on various target platforms. The platform on which the CodeWarrior runs is called the "host." From that host, the IDE is used to develop code to "target" various platforms.

This chapter assumes familiarity with the CodeWarrior Integrated Development Environment (IDE). Refer to the *CodeWarrior IDE User Guide* for any questions related to the IDE.

## Creating a new project

A project contains one or more build targets. Each build target in a project contains a collection of files that the IDE uses to build a output file. Build targets within a project may share some or all of their files. Some examples of a output file include an application, static library, or dynamic library.

Each build target within a project has its own options that customize how the IDE builds the output file. There are a wide variety of options that control code optimization, browsing, debugging, compiler warnings, and much more.

This section discusses the basic tasks involving in creating a simple project called “testing” . It included activities such as creating a project, opening the project, adding a file, and saving a project. For more advance operations such as moving files in the Project window, marking files for debugging, creating nested projects and build targets, and dividing the Project window into groups of files, please refer to the Metrowerk CodeWarrior IDE user guide.

Creating a new CodeWarrior project file involves setting up the following:

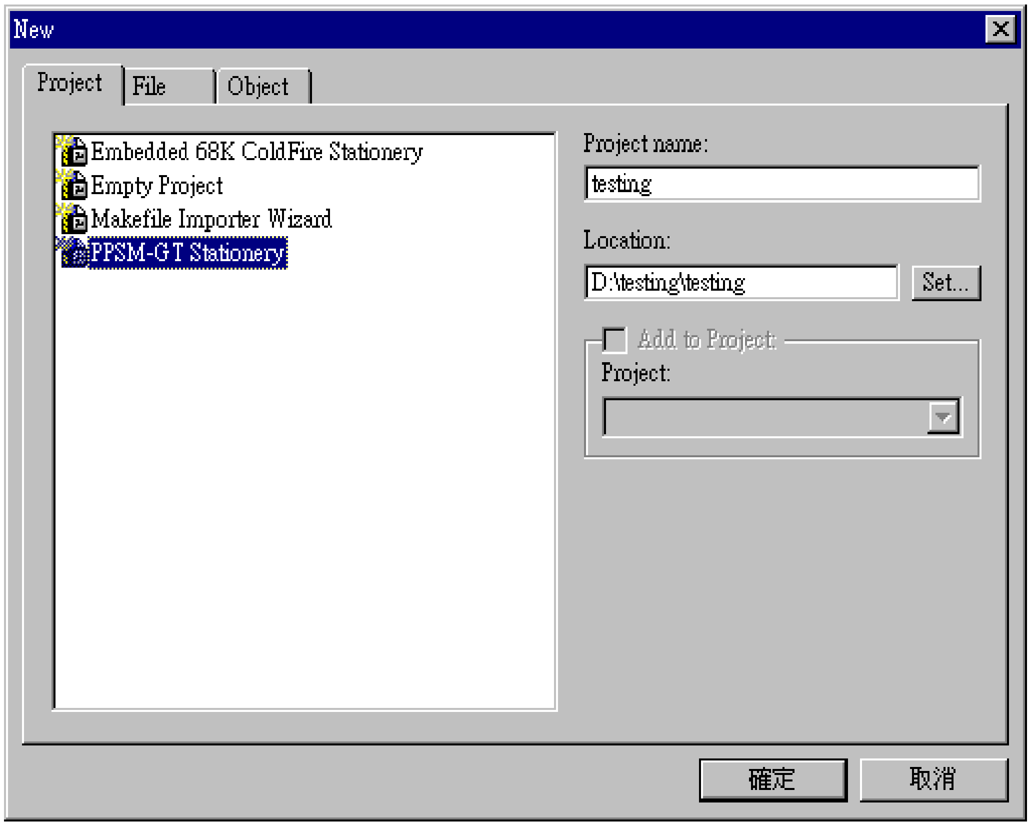
- Types of Project Files
- Choosing a Project Stationery File
- Naming Your New Project

For the simple project, the following are the project setting and as shown in Figure 5.1 as followed:

- The Project’s Type is a Project Stationery Type that consists of pre-configured libraries, source code placeholders, and resource file placeholders.

- The Project’s Stationary selected is PPSM-GT Stationary
- The Project’s Name is “testing”

**Figure 5.1 Selecting Embedded 68K Stationery**



Embedded 68K ColdFire Stationery
Empty Project
Makefile Importer Wizard
PPSM-GT Stationery

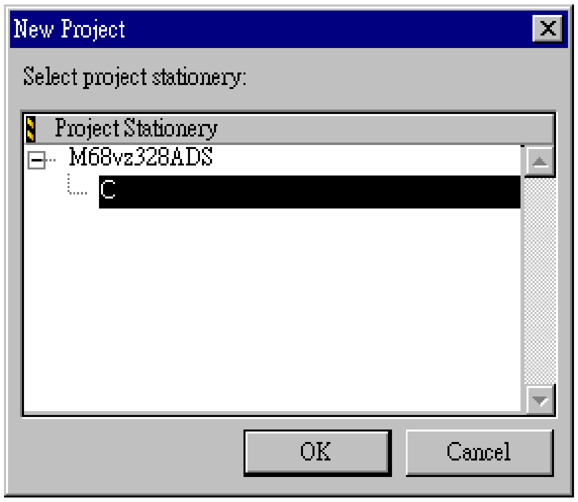
Project name: testing

Location: D:\testing\testing

Next the system will prompt the developers to choose the language used for the development. C language is selected for the development of the application.

In the respective directory under the Project Stationery, choose the target ADS that is being used. Select ADS_68VZ328 as shown in Figure 5.2, and click OK.

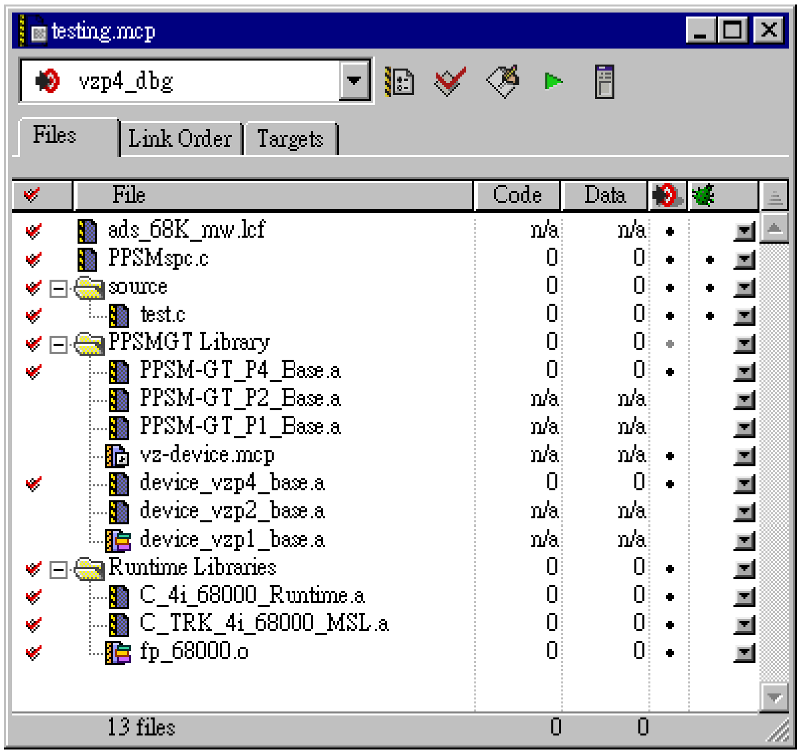
**Figure 5.2 Selecting the related device**



The project will be generated. The appropriate runtime libraries, linker command file (lcf), and a main source file with the "Hello World" application are included. Runtime libraries include as shown in Figure 5.3:

- C_4i_68000_Runtime.a for the general 68K runtime support,
- C_TRK_4i_68000_MSL.a for all the MetroTRK debug monitor console I/O and Metrowerks Standard Library support, and
- fp_68000.o for 68K floating point support

**Figure 5.3 Runtime Libraries**



testing.mcp

vzp4_dbg

Files | Link Order | Targets

| ✓ | File | Code | Data | | | |
|---|---|---|---|---|---|---|
| ✓ | ads_68K_mw.lcf | n/a | n/a | • | | |
| ✓ | PPSMspc.c | 0 | 0 | • | • | |
| ✓ | source | 0 | 0 | • | • | |
| ✓ | test.c | 0 | 0 | • | • | |
| ✓ | PPSMGT Library | 0 | 0 | • | | |
| ✓ | PPSM-GT_P4_Base.a | 0 | 0 | • | | |
| | PPSM-GT_P2_Base.a | n/a | n/a | | | |
| | PPSM-GT_P1_Base.a | n/a | n/a | | | |
| | vz-device.mcp | n/a | n/a | • | | |
| ✓ | device_vzp4_base.a | 0 | 0 | • | | |
| | device_vzp2_base.a | n/a | n/a | | | |
| | device_vzp1_base.a | n/a | n/a | | | |
| ✓ | Runtime Libraries | 0 | 0 | • | | |
| ✓ | C_4i_68000_Runtime.a | 0 | 0 | • | | |
| ✓ | C_TRK_4i_68000_MSL.a | 0 | 0 | • | | |
| ✓ | fp_68000.o | 0 | 0 | • | | |
| | 13 files | 0 | 0 | | | |

The simple example “Hello World” is provided to assist developers understand the structure of PPSM-GT program and get started with the minmium confguration. Developers can modify the main program file and the lcf file to suit their application. More files can be added to the project when required.

## Linker Command File

The default lcf file created is shown as below:

**Listing 5.1 Default lcf file**

```
# Sample Linker Command File for Metrowerks
Embedded 68K/ColdFire

MEMORY {
    TEXT (RWX) : ORIGIN = 0x00008000, LENGTH =
0x00000000
```

```
     DATA (RW) : ORIGIN = AFTER(TEXT), LENGTH =
0x00000000
}

SECTIONS {
     .main_application :
     {
       *(.text)
       .= ALIGN(0x4);
       *(.rodata)
     } > TEXT

    .main_application_data :
     {
       . = ALIGN(0x4);
       *(.exception)
       . = ALIGN(0x4);
       __exception_table_start__ = .;
       EXCEPTION
       __exception_table_end__ = .;

       .= ALIGN(0x4);
       ___sinit__ = .;
         STATICINIT

       .= ALIGN(0x4);
       __START_DATA = .;
       *(.data)
       __END_DATA = .;

       .= ALIGN(0x4);
       __START_SDATA = .;
       *(.sdata)
       __END_SDATA = .;

       .= ALIGN(0x4);
       __SDA_BASE = .;# A5 set to  middle of data
and bss ...

       .= ALIGN(0x4);
       __START_SBSS = .;
       *(.sbss)
```

```
          *(SCOMMON)
          __END_SBSS = .;

          .= ALIGN(0x4);
          __START_BSS = .;
          *(.bss)
          *(COMMON)
          __END_BSS = .;

          . = ALIGN(0x4);
        } > DATA

        ___SP_END = .;
        __SP_INIT = . + 0x00004000;# set stack to
0x4000 bytes (16KB)
        __TRK_SP_INIT = __SP_INIT;
        _startof_bss = __START_BSS;
        ___heap_addr = __SP_INIT;# heap grows in
opposite direction of stack
        ___heap_size = 0x400000;# heap size set to
0x400000 bytes (500KB)

        _sizeof_bss = __END_BSS - __START_BSS;
        _sizeof_data = __END_DATA - __START_DATA;

        __S_romp = 0x0;# no ROM in this example
}
```

For detail specification of the lcf file format, please refer to Metrowerks document at:

Metrowerks / CodeWarrior / Documentation / CodeWarrior / HTML / Targeting_Embedded_68K_html / 68K072_ELFLinker.fm.html#489147

The above lcf file has to be modified to add several linker flag including LCD physical and virtual size and the UART receive buffer size in order for it to work properly with LCD display. These settings can be added at the end of the lcf file as shown below:

## Saving a Project

The CodeWarrior IDE automatically updates and saves your project when you perform certain actions. Some of the actions that cause a project file to get saved are as followed:

- Close the project
- Change Preferences or Target Settings for the project
- Add or delete files for the project
- Compile any file in the project
- Edit groups in the project
- Remove object code from the project
- Quit the CodeWarrior IDE

When the CodeWarrior IDE automatically saves the project, it saves the following information:

- The names of the files added to your project and their locations
- All configuration options
- Dependency information (such as the touch state and interface file lists)
- Browser information
- The object code of any compiled source code files

## Choosing the Target Setting

This section discusses setting options for individual build targets. These options configure the IDE to suit the needs when building the targets in the project.

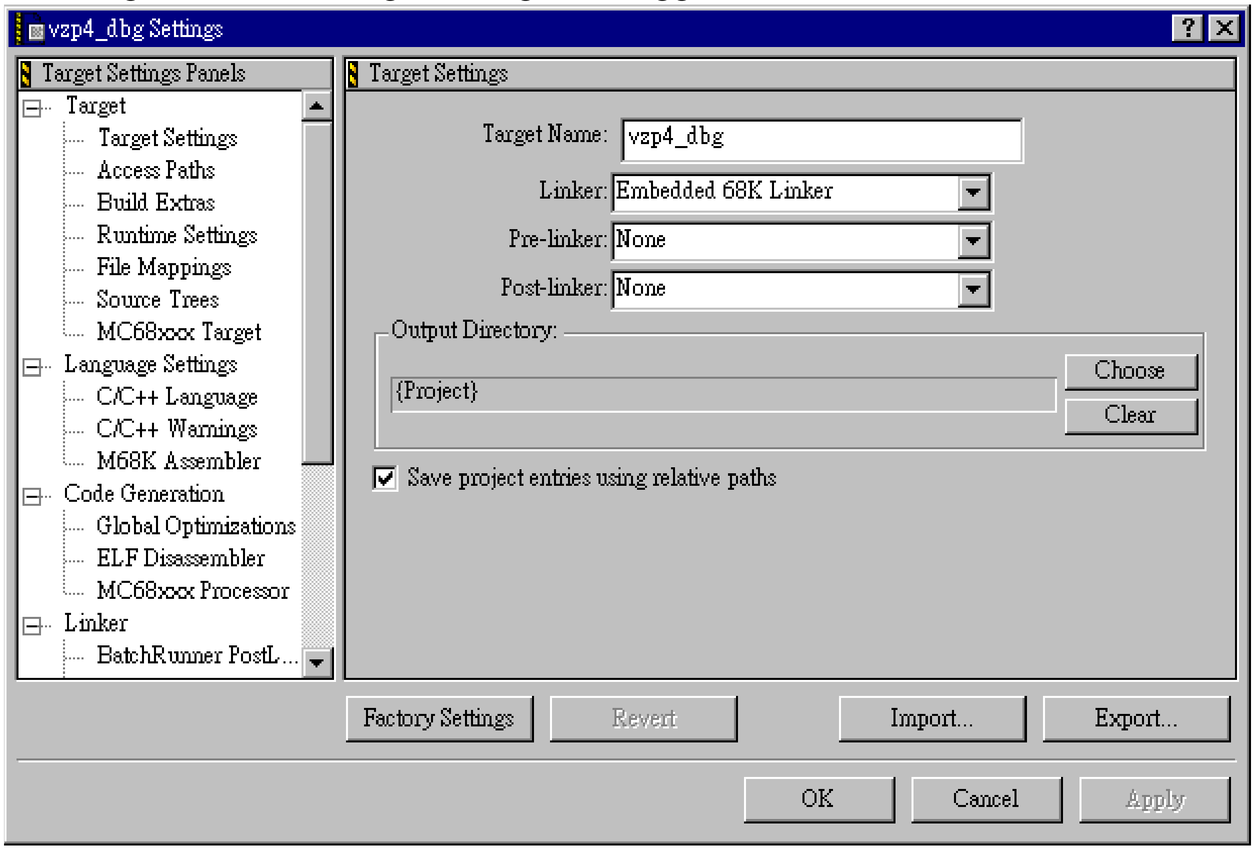
A CodeWarrior project can contain one or more build targets. A build target contains all the information required to identify which files belong in a particular build, the complier and linker setting for the build, the output information, and so on.

To view the target seeting panel, in Figure 5.3 click on the “Target” label and a list of current supported target will be shown. To view the setting for each target, double click on the selected target and a new window like Figure 5.4 will be displayed. This is the target setting window and the setting panels that appear in the Target Settings window depend on the operating system or chip family of the build target and the programming language used.

In the simple project the target hardware is the dragonball 68VZ328 and C is the programming language as shown in Figure 5.4

New target setting could be created to suit the needs of the development. Refer to the Metrowerk CodeWarrior IDE user guide.

**Figure 5.4 Setting the Target Debugger**

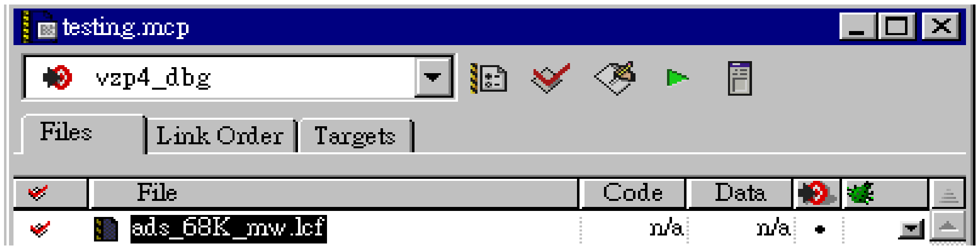


Target Name: vzp4_dbg
Linker: Embedded 68K Linker
Pre-linker: None
Post-linker: None
Output Directory: {Project}
[✓] Save project entries using relative paths

## Making and Running the Program

Select the desired target and then click on the "Make" icon as shown in Figure 5.5.

**Figure 5.5 Making the program**



After successful compilation and link, all the red check marks on the left-most column will disappear, indication a successful update. The program will be generated in the specified location.

With the "Connection Settings" in the project settings mentioned in the previous section properly configured and the physical serial connection to the ADS properly set up, press the "Run" icon and download the application to the ADS and debug the program.

After successfully compiling this project, the output file can be downloaded to the actual hardware—for example, the MC68VZ328ADS board—if it is connected to the host machine. The connection settings might need to be changed under the Debugger / Connection Setting option in the project setting panel.

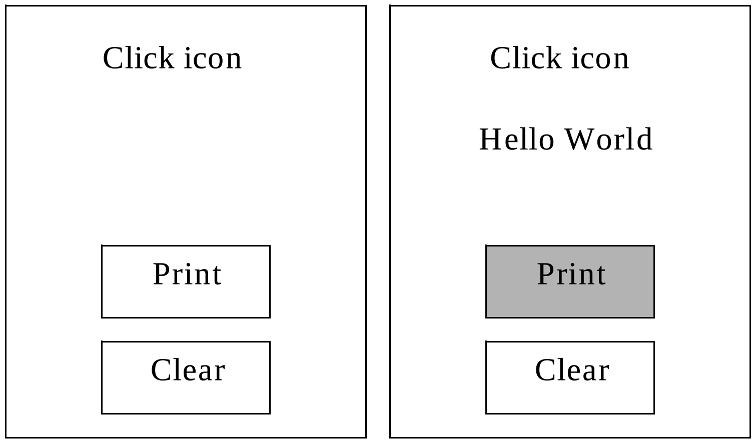
## Building An PPSM-GT Application

This section provides a step by step guidances for building an application using PPSM-GT library with the smallest possible PPSM-GT application “Hello World”. The application specification is as specified in Application specification, and could be used as the building block for more complex application.

### Application specification

The application is to display a bitmap and a string on the LCD screen as shown in the Figure 5.6 . As the icon button on the touch screen is touched, the icon button will be inversed and the text “Hello World” will be displayed.

**Figure 5.6 Display of the sample program as the icon is dragged/ undragged .**



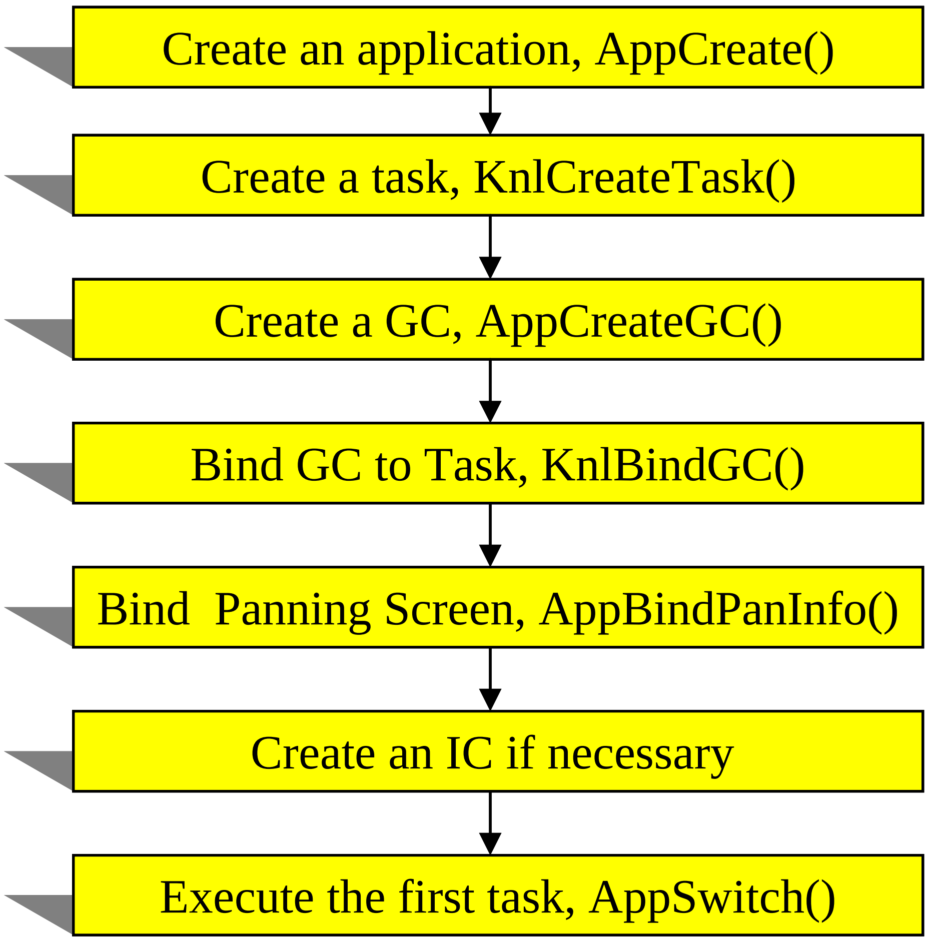
Designing the "Hello World Application

## Designing the “Hello World” application.

The “Hello World” is designed and developed using the typical flow for creating a PPSM-GT application as shown in Figure 5.7. There are seven steps in this typical flow and the steps varies with the complexity of the application. For example, more tasks, graphic contexts, and input contexts might need to be created to support more complex application. In addition, multiple applications may also be required instead of a simple application as shown in this example.

The code example of the “Hello World” application is attached for reference in Source Code Of Hello World

**Figure 5.7 Typical Flow for creating an Application with a Task an Panning Screen**


Create an application, AppCreate()
Create a task, KnlCreateTask()
Create a GC, AppCreateGC()
Bind GC to Task, KnlBindGC()
Bind Panning Screen, AppBindPanInfo()
Create an IC if necessary
Execute the first task, AppSwitch()

### Create an application using AppCreate( )

Inside the main routine which is the entry point of the application program. User has to create their applications by using the below API.

```
 /* To create an application */
 if (AppCreate(&appId1, name1, 0, 0, 0, 0, 0,
3000) != SYS_OK)
 {
    return(SYS_ERR);
 }
```

### Create a task using KnlCreateTask( )

Then user has to create the corresponding task for the application.

```
   /* To create a task */
if (KnlCreateTask(&taskId01, (P_VOID)Test, name1,
3000, 1, SUPERVISOR_MODE) != SYS_OK)
{
     return(SYS_ERR);
}
```

### Create a graphic context using GpxCreateGC()

User can create their Graphic Context with the a panning screen attached.

```
 /* To create graphic context with a panning
screen */
AppCreateGC(&gcId1, 160, 240);
```

### Bind the Task to Graphic Context with KnlBindGC()

The created Graphic Context should attached to the corresponding task where the helloWorldId is the ID number of the task and helloGCId is the ID number of the created Graphic Context.

```
 /* To bind a graphic context ID which is created
before to the HelloWorld task.*/
if (KnlBindGC(taskId01, gcId1) != SYS_OK)
{
    return(SYS_ERR);
}
```

### Bind App to Panning Screen with AppBindPanInfo()

User can obtain the panning screen ID number by the API GpxGetPaScreen. With the panning screen ID number, user can attach it to the application.

```
 /* bind the same panning screen and panning info.
to the application */
```

```
if (GpxGetPanScreen(gcId1, &screen1) != SYS_OK)
{
    return(SYS_ERR);
}
if (AppBindPanInfo(appId1, screen1) != SYS_OK)
{
    return(SYS_ERR);
}
```

### Create IC for helloWorld task

User can create their Input Contexts and active areas and make initialization like the below codes:

```
/* Create IC for helloWorld task */
    PenCreateIC(&gPenICId);
     /* To init the input context */
     PenInitIC(gPenICId, taskId01, 0, PEN_32HZ, 1,
1, BLACK);
     /* Set ring buffer size for pen event handling
*/
     PenSetRingBuffer(appId1, 6000);
     /* To insert the input context to the front of
an input context
       list under the application context */
     AppAddIC(appId1, gPenICId);
```

### Start the task with AppSwitch()

With all the stuffs created, user can start running the task by the API AppSwitch. For a usual PDA application, the first task should be a menu task.

```
/* Switch to the helloWorld application */
AppSwitch(appId1);
```

### Inside the first task routine

Inside the first calling task, the user should initialize the LCD screen and make calibration to the input pen if the hardware is present. Like most typical PPSM-GT task, there is a event wait loop at the end of the routine to wait for event for that task. Figure 5.8 shown the typical task flow chart.

To enhance the understanding, the codes below is provided to highlight the events performed by the task:

```
    /* Clear the screen and perform input pan
calibration*/
    GpxFillScreen(WHITE);
    PenCalibrate(TRUE);
```

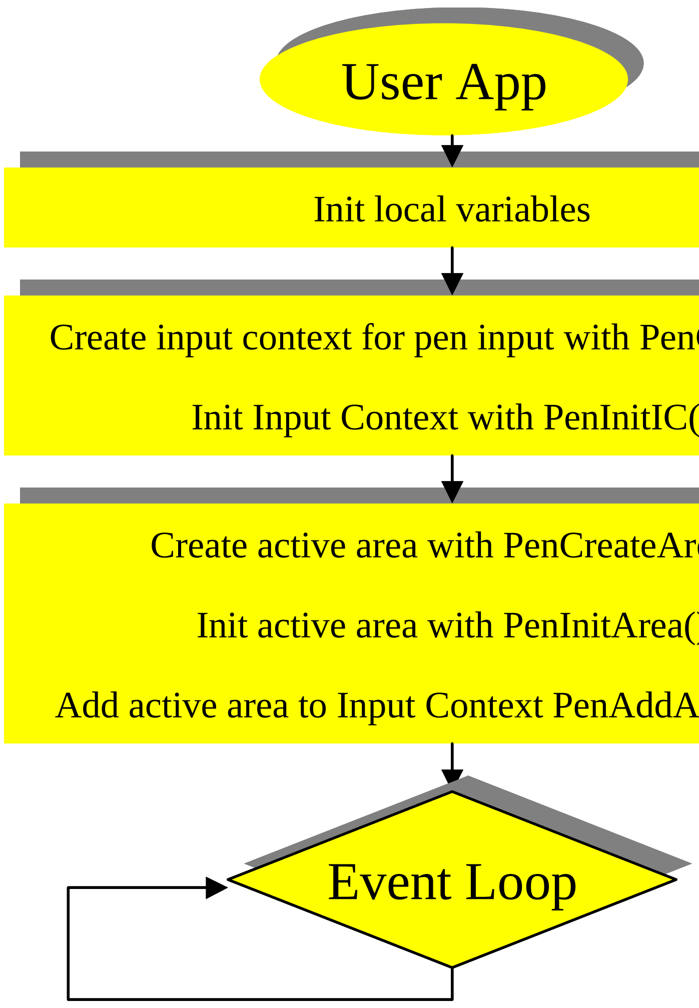
Then user can put their own codes to create those necessary resources and make initialization.

Finally, user should add an event get loop to grasp the input event for the corresponding event processing. For example, the events could be EVT_PEN_ICON_DRAG, EVT_PEN_ICON_DRAG_OUT, EVT_PEN_ICON_TOUCH, EVT_PEN_ICON_UP ... etc.

```
/* Event Get Loop to check the event for
processing */
/* Get interrupt */
    while (1)
    {
        switch(EvtGet())
        {
            case EVT_PEN_ICON_DRAG:
            case EVT_PEN_ICON_DRAG_OUT:
            case EVT_PEN_ICON_TOUCH:
                pEvent = EvtGetEvent();
                PenGetAreaIdFromEvent(pEvent,
&areaId);
              PenGetAreaPos(areaId, &xSrc, &ySrc,
&xDest, &yDest);
                GpxInvRec(xSrc, ySrc, xDest,
yDest);
                break;

            case EVT_PEN_ICON_UP:
             /* Free the memory occupied by the
event */
                pEvent = EvtGetEvent();
```

```
          PenGetAreaIdFromEvent(pEvent, &areaId);
              PenGetAreaPos(areaId, &xSrc, &ySrc,
&xDest, &yDest);
          GpxInvRec(xSrc, ySrc, xDest, yDest);

          if(areaId == printId)
        {
                    /* print hello world */
                    Typing(DEFAULT_FONT,
REPLACE_STYLE, BLACK, STR_XSRC, STR_YSRC, 16,
(P_TEXT)helloWorldTxt);
          }
          else if(areaId == clearId)
        {
                    /* Clear hello */
                    GpxSetColor(WHITE);
                    GpxSetStyle(REPLACE_STYLE);
                   GpxFillRec(STR_XSRC, STR_YSRC,
STR_XDEST, STR_YDEST);
          }

      default:
          break;
        }
    }
```

**Figure 5.8 Typical flow for User routine to handle pen inputs**


User App
Init local variables
Create input context for pen input with PenCreateIC()
Init Input Context with PenInitIC()
Create active area with PenCreateArea()
Init active area with PenInitArea()
Add active area to Input Context PenAddAreaToIC()
Event Loop

## Summary

This chapter explains the two development environments that the PPSM-GT is using. It also provide basic information on developing a simple PPSM-GT application.

## Code Examples

**Listing 5.2 Hello world makefile**

```
#######################################
#
# Makefile for HelloWorld
#
######################################

RES_DIR=.

NAME = Sample

SYSDIR = c:\sds74\lib68000\include
LIBDIR = c:\sds74\lib68000
PPSMDIR = c:\ppsmgt\
PPSMEZLIB = c:\ppsmgt\libvz328
PPSMDEV = c:\ppsmgt\device\vz328\skeldev

OBJSFILE = objs # indirect obj file name

INCDIR = $(PPSMDIR)\INCLUDE
SRCDIR = .\SOURCE
OBJDIR = .\OBJ

DEFAULT =

OBJ = $(OBJDIR)\test.o

PPSMLIB = $(PPSMEZLIB)\ppsm2.a $(PPSMDEV)\skeldev2.a

DEF = $(DEFAULT) -I$(INCDIR) -I .\INCLUDE
CCOPT = -f -Og -DPIXEL_2
ASOPT = -f

CC    = cc68000 $(CCOPT)
ASM = as68000 -V 68000 -I$(SYSDIR) -I$(INCDIR) -f
LINK  = linker -y

#Download version
$(NAME).out: $(OBJ) $(NAME).spc $(PPSMLIB)
```

```
$(LINK) -F objs -E errs -o $(NAME).out -f $(NAME).spc

# C files
$(OBJDIR)\Sample.o: $(SRCDIR)\Sample.c $(INCDIR)\ppsm.h $(CC)
$(SRCDIR)\Sample.c -E errs $(DEF) -o object=$*.o
```

**Listing 5.3 Source Code Of Hello World**

```
/
*****************************************************************
********/
This is the source code of a sample application. There are a total
of 2 files provided in the appendix test.c and test.h. The files
could also be found in the sample directory in the PPSM-GT CDROM.
This appendix is intended to clarify your understanding in
building an application with PPSM-GT. You can use this sample
program as a starting point for your application.

/
*****************************************************************
********/
/*                              test.c
*/
/
*****************************************************************
***************

 C   M O D U L E   F I L E

 (c) Copyright Motorola Semiconductors Hong Kong Limited 2000-
2001
 ALL RIGHTS RESERVED

*****************************************************************
****************

 Project Name : Portable Personal System Manager GT version 1.1
 Project No.  : PDAPSM04
 Title        :
 File Name    : test.c
 Last Modified: May 29, 2001

 Description  : This is a very simple sample program to demostrate
the procedure
                to write a PPSM-GT program. The program will
display 2 text icons,
               when user selects the Print icon, "Hello World" is
displayed on
              LCD, when Clear icon is selected, the "Hello World"
string will be
                cleared.

 Cautions     : NA.

*****************************************************************
****************/

#include <OS.H>

#include "test.h"




main(void)
{
    TASK_ID     taskId01;
    APP_ID      appId1;
    GC_ID       gcId1;
    SCREEN_ID   screen1;
    STATUS      status;
    TEXT        name1[] = {’T’, ’e’, ’s’, ’t’, ’1’, 0};

  /* To create an application */
  if (AppCreate(&appId1, name1, 0, 0, 0, 0, 0, 3000) != SYS_OK)
  {
    return(SYS_ERR);
  }

  /* To create a task */
  if (KnlCreateTask(&taskId01, (P_VOID)Test, name1, 3000, 1,
SUPERVISOR_MODE) != SYS_OK)
```

```
  {
    return(SYS_ERR);
  }

  /* To create graphic context with a panning screen */
  AppCreateGC(&gcId1, 160, 240);


  /* To bind a graphic context ID which is created before to the
HelloWorld task.*/
  if (KnlBindGC(taskId01, gcId1) != SYS_OK)
  {
      return(SYS_ERR);
  }

  /* bind the same panning screen and panning info. to the
application */
  if (GpxGetPanScreen(gcId1, &screen1) != SYS_OK)
  {
      return(SYS_ERR);
  }

  if (AppBindPanInfo(appId1, screen1) != SYS_OK)
  {
      return(SYS_ERR);
  }

    PenSetRingBuffer(appId1, 6000);
    PenCreateIC(&gPenICId);
    PenInitIC(gPenICId, taskId01, 0, PEN_32HZ, 1, 1, BLACK);
    AppAddIC(appId1, gPenICId);

  /* Switch to the helloWorld application */
  AppSwitch(appId1);

    return;
}

void  Test(void)
{
/* Active area Id */
U32     printId, clearId;
```

```
/* variable for EvtGet() */
AREA_ID         areaId;
P_EVENT         pEvent;
S16             xSrc, ySrc, xDest, yDest;

    GpxFillScreen(WHITE);
    PenCalibrate(TRUE);

    /* print title */
    Typing(DEFAULT_FONT, REPLACE_STYLE, BLACK, TITLE_XSRC,
TITLE_YSRC, 16, (P_TEXT)titleTxt);

    /* Create a button to print Hello World */
    TextIcon((P_U32)&printId, gPenICId, ICON_XSRC, ICON_YSRC,
ICON_WT, ICON_HT,
        DEFAULT_FONT,(P_TEXT)printTxt);

    /* Create a button to clear Hello World */
    TextIcon((P_U32)&clearId, gPenICId, ICON_XSRC,
ICON_YSRC+ICON_HT+ICON_OFFSET, ICON_WT, ICON_HT,
        DEFAULT_FONT,(P_TEXT)clearTxt);

    /* Get interrupt */
    while (1)
    {
        switch(EvtGet())
        {
            case EVT_PEN_ICON_DRAG:
            case EVT_PEN_ICON_DRAG_OUT:
            case EVT_PEN_ICON_TOUCH:
                pEvent = EvtGetEvent();
                PenGetAreaIdFromEvent(pEvent, &areaId);
                PenGetAreaPos(areaId, &xSrc, &ySrc, &xDest,
&yDest);
               GpxInvRec(xSrc, ySrc, xDest, yDest);
                break;

            case EVT_PEN_ICON_UP:
          /* Free the memory occupied by the event */
                pEvent = EvtGetEvent();
        PenGetAreaIdFromEvent(pEvent, &areaId);
```

```
                PenGetAreaPos(areaId, &xSrc, &ySrc, &xDest,
&yDest);
        GpxInvRec(xSrc, ySrc, xDest, yDest);

          if(areaId == printId)
        {
                    /* print hello world */
                    Typing(DEFAULT_FONT, REPLACE_STYLE, BLACK,
STR_XSRC, STR_YSRC, 16, (P_TEXT)helloWorldTxt);
        }
        else if(areaId == clearId)
        {
                    /* Clear hello */
                    GpxSetColor(WHITE);
                    GpxSetStyle(REPLACE_STYLE);
                    GpxFillRec(STR_XSRC, STR_YSRC, STR_XDEST,
STR_YDEST);
        }

    default:
        break;
        }
    }
}


U16 Strlen(P_TEXT str)
{
    U16 len=0;

    while (str[len])
        len++;
    return len;
}

/*
 *   Prints out message(a row only) on the screen start at (xSrc,
ySrc)
 */
void Typing(U8 font, U8 style, U8 greylev, U16 xSrc, U16 ySrc, U8
bitLen, P_TEXT str)
{
    U16 len;
    U32 tId;

    /*  create the text template  */
    TxtCreateTmplt(&tId);

    /*  find out the length of the message and print it out  */

    if(bitLen == 8)
    {
        if (len = strlen((P_U8)str))
        {
            TxtSetupTmplt(tId, font, style, greylev, xSrc, ySrc,
len, 1);
            TxtMap(tId, EIGHT_BIT, (P_U8)str, len);
        }
    }
    else
    {
        if (len = Strlen(str))
        {
            TxtSetupTmplt(tId, font, style, greylev, xSrc, ySrc,
len, 1);
            TxtMap(tId, SIXTEEN_BIT, (P_TEXT)str, len);
        }
    }

    TxtDeleteTmplt(tId);
}


/* Create an icon with text */
STATUS TextIcon(P_U32 areaId,IC_ID icId,U16 xSrc,U16 ySrc,U16
width,U16 height,U16 font, P_TEXT message)
{

    U16     xDest,yDest;
    U16     x, y, sizex, sizey;
    U16     len;
    U32     tId;
    STATUS  ret;
```

```
#ifdef PIXEL_4
    COLOR  light = GREY6;
    COLOR  dark = GREY10;
#elif defined(PIXEL_2)
    COLOR  light = LIGHT_GREY;
    COLOR  dark = DARK_GREY;
#else
    COLOR  light = WHITE;
    COLOR  dark = BLACK;
#endif

    xDest = xSrc + width;
    yDest = ySrc + height;

    if ((xDest > GpxGetDisplayWidth()) || (yDest >
GpxGetDisplayHeight())) return SYS_ERR;

    PenCreateArea(areaId);
    PenInitArea(*areaId, xSrc, ySrc, xDest, yDest, ICON_AREA, 0,
0, 1);

  ret=PenAddAreaToIC(icId, *areaId);
    GpxSetColor(BLACK);
    GpxSetStyle(REPLACE_STYLE);
    ret=GpxDrawRec(xSrc,ySrc,xDest,yDest,0);
    ret=GpxDrawRec(xSrc+3,ySrc+3,xDest-3,yDest-3,0);

  GpxSetColor(light);
    GpxSetColor(BLACK);
    ret=GpxDrawLine(xSrc+1,ySrc+1,xSrc+2,ySrc+2,0);

  GpxSetColor(dark);
  GpxSetStyle(REPLACE_STYLE);
    ret=GpxDrawLine(xSrc+1, yDest-1, xDest-1, yDest-1, 0);
    ret=GpxDrawLine(xDest-1, ySrc+1, xDest-1, yDest-1, 0);
    ret=GpxDrawLine(xSrc+2, yDest-2, xDest-2, yDest-2, 0);
    ret=GpxDrawLine(xDest-2, ySrc+2, xDest-2, yDest-2, 0);

    TxtCreateTmplt(&tId);
    len = Strlen(message);
    if ((font==SMALL_NORMAL_FONT) || (font==SMALL_ITALIC_FONT))
    {
```

```
    sizex = 8;
    sizey = 8;
  }
  else
  {
    sizex = 16;
    sizey = 16;
  }

    /* Calculate the co-ordinate for fonts */
    x=xSrc+((xDest-xSrc)-len*sizex)/2+2;
    y=ySrc+((yDest-ySrc)-sizey)/2+2;

    TxtSetupTmplt(tId,font,OR_STYLE,BLACK,x,y,len,1);
    TxtMap(tId, SIXTEEN_BIT, (P_TEXT)message, len);
    TxtDeleteTmplt(tId);

    return SYS_OK;
}   /* TextIcon */

/* Icon coordinate */
#define TITLE_XSRC  35
#define TITLE_YSRC  10

#define ICON_XSRC   45
#define ICON_YSRC   130
#define ICON_WT     70
#define ICON_HT     24
#define ICON_OFFSET 10

#define STR_XSRC    33
#define STR_YSRC    60
#define STR_XDEST   128
#define STR_YDEST   69

#define DEFAULT_FONT   SMALL_NORMAL_FONT

// TEST.H
const   TEXT    titleTxt[]={’C’,’l’,’i’,’c’,’k’,’
’,’i’,’c’,’o’,’n’,0};
const   TEXT    printTxt[]={’P’,’r’,’i’,’n’,’t’,0};
const   TEXT    clearTxt[]={’C’,’l’,’e’,’a’,’r’,0};
const   TEXT    helloWorldTxt[]={'H','e','l','l','o','
','W','o','r','l','d',0};

// Function prototype
void  Test(void);
U16 Strlen(P_TEXT str);
void Typing(U8 font, U8 style, U8 greylev, U16 xSrc, U16 ySrc, U8
bitLen, P_TEXT str);
STATUS TextIcon(P_U32 areaId,IC_ID icId,U16 xSrc,U16 ySrc,U16
width,U16 height,U16 font, P_TEXT message);

// Global variable
IC_ID   gPenICId;
```

# 6 PPSM-GT Configuration Mechanism

PPSM-GT supports variant target hardware configuration and third-party fonts through modifying its device drivers. The purpose is to allow developers and designers to customized PPSM-GT to suit their hardware. All the files mentioned in the following sections are stored in the device driver directory and are provided for customization. PPSM-GT provide the system call and developers and designers have to provide the content of the routoines.

These routines are included to allow the developers and designers to better customize the operation of the various devices. This improves flexibility, helping users to customize their systems.

Defaults values are often provided as a reference and developers and designer can use them as it is or modified them to their hardware needs. The following are the type of device driver available:

- Font driver(font.c)
- LCD Device Driver (lcddev.s)
- Pen input driver
- System bootup driver (boot.s)
- System interrupt handling drivers
- System power control driver
- IrDA driver

# Font driver(font.c)

The font driver provides PPSM-GT developers and designers to support multiple font libraries. PPSM-GT support a set of default fonts as shown in Listing 6.2. Developers and designers could add to the font selection if they are able to obtain other fonts. The structure of the font driver consists of 2 elements: a data structure and two functions.

## Font Driver Structure

A data structure type FONTLIB is required to store information about the font libraries being used.

The FONTLIB type is defined as follow:

**Listing 6.1 Font structure**

```
typedef struct
{
  P_U8baseAddr;
  U16fontType;
  U16fontWidth;
  U16fontHeight;
  U16bitmapSize;
} 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:

**Listing 6.2 Default PPSM Font library**

```
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},
  };
```

The fontLib data structure above are indexed into by the corresponding PPSM-GT font types.

## Font Library or Font Generation Engine Initialization

| | |
|---|---|
| void **FontInit**(void) | 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.

## Font Accessing

| | |
|---|---|
| P_U8 **FontGetCharAddr**(P_FONTATTR *pFont*, TEXT *code*) | This function returns the font bitmap of a character based on the given font attributes and character code.<br><br>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.<br><br>Since PPSM-GT 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:<br>• SMALL_NORMAL_FONT<br>• SMALL_ITALIC_FONT<br>• LARGE_NORMAL_FONT<br>• LARGE_ITALIC_FONT |
| **Parameters** | |
| *pFont* | pointer to a FONTATTR structure which describes the font |
| *code* | character code for which the font lookup or generation is performed |
| **Return** | |
| N/A | pointer to the bitmap of the character specified by the given character code (the bitmap is represented by unsigned 8-bit values |

# LCD Device Driver (lcddev.s)

PPSM-GT allows developers to customize the LCD Driver according to the developers' preference. The file that contains the LCD information is lcddev.s.

Three 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. Users **must modify them** for their LCD panel.

## 1 bit/pixel Initialization

| | |
|---|---|
| void **_LCDDev1**(void) | This function initializes the LCD controller for 1 bit / pixel graphics mode. Application programmers may add whatever statement to initialize the LCD module. |

## 2 bits/pixel Initialization

| | |
|---|---|
| void **_LCDDev2**(void) | This function initializes the LCD controller for 2 bits / pixel graphics mode. Application programmers may add whatever statement to initialize the LCD module. |

## 4 bits/pixel Initialization

| | |
|---|---|
| void **_LCDDev4**(void) | This function initializes the LCD controller for 4 bits / pixel graphics mode. Application programmers may add whatever statement to initialize the LCD module. |

Please refer to the appropriate integrated processor user's manual for information on the LCD controller and the registers' definitions.

# Pen input driver

There are two part of the pen input driver.

- Pen device driver—It reads the raw coordinate data from the touch panel through the A / D convertor and SPI. There are four functions in this driver. They are Pen Initialization, Pen Interrupt Enable, Pen Interrupt Disable and Pen Read Device.
- Pen calibration driver—It calibrates the touch panel and converts the raw coordinate data to LCD coordinate data.

Users **must modify the pen device driver** for their A / D convertor.

Users **must modify the pen calibration driver if** their touch panel is very different from the one that comes with ADS broad.

## Pen device driver

The pen device driver is mainly constructed by the two files pendev.c and pendev.h.

### pendev.h

It is the header file for pendev.c.

### pendev.c

The pendev.c file performs the following functions: Pen Initialization, Pen Interrupt Enable, Pen Interrupt Disable and Pen Read Device.

| | |
|---|---|
| void **PenDevInit**(void) | This function initializes SPI and all the ports that are used for pen sampling. |
| void **PenIrptEnable**(void) | This function enables the Pen Interrupt, $\overline{\text{PENIRQ}}$, for pen down detection. |

| | |
|---|---|
| void **PenIrptDisable**(void) | This function disables the Pen Interrupt, $\overline{\text{PENIRQ}}$. |
| STATUS **PenReadDevice**(P_S16 x, P_S16 y) | This function reads the raw coordinate from the touch panel through the A/D convertor and SPI. |

## Pen calibration driver

The Pen calibration driver mainly performs the following functions: PutLogo, CrossSetup, ConvertLCD, ConvertLCDX, ConvertLCDY, CalibratePen and SetFactors. The driver is constructed by the two files peninit.h and peninit.c.

### peninit.h

It is the header file for peninit.c.

### peninit.c

| | |
|---|---|
| void PutLogo() | This function is used to display the Motorola logo on the LCD display. |
| static void CrossSetup(void) | Draw a set of crosses on the screen for calibration purpose. |
| static S16 ConvertLCD( U16 value, U32 factor, U32 offset) | Convert touch panel co-ordinate to LCD coordinates. |
| S16 ConvertLCDX( U16 value) | Convert touch panel x co-ordinate to LCD x coordinates. |
| S16 ConvertLCDY( U16 value) | Convert touch panel y co-ordinate to LCD y coordinates. |
| STATUS CalibratePen( U8 logoFlag) | Calibrate the touch panel with the screen. This routine prints out two crosses, one at the top-right corner and the other at the bottom-left corner. The user must press on the crosses to start the system. |
| STATUS SetFactors() | Set touch panel to LCD coordinate scaling factor. |

## System bootup driver (boot.s)

The system boot strap code is responsible for initializing the chip's internal devices and mapping chip-selects for ROM and RAM of the hardware system at boot time. Different hardware memory configurations and systems require different boot strap code. An example boot strap code is included in the PPSM-GT device driver library to demonstrate how to boot strap and initialize the chip-selects for the MC68VZ328ADS system.

## System interrupt handling drivers

The interrupt handler device driver allows users to install their own interrupt handlers for certain kinds of interrupts.

Users can add additional interrupt handlers besides the default handler provided by the system in the device driver to perform exception handling. Since PPSM-GT does not manage or monitor these user-defined handlers, be careful when installing them.

There are six handlers that can be used. They are DevIrpt6Handler, DevIrpt5Handler, DevIrpt4Handler, DevIrpt3Handler, DevIrpt2Handler and DevIrpt1Handler.

A single argument is passed into the interrupt handler. This argument is the value of the interrupt status register.

## System power control driver

The system power control driver contains the following functions: PortEnterIdle, PortExitIdle, PortEnterDoze, PortExitDoze, PortEnterSleep and PortExitSleep.

The functions in this driver are for controlling devices when going into and out of Normal, Idle, Doze, and Sleep mode. Users must add their own device-controlling routines if any of the devices are used in a system that needs to be switched on/off when going into/out of these modes.

## IrDA driver

To support different manufacturers' IrDA transceivers, PPSM-GT provides the function calls shown in Table 6.1 to enable developers to program the IrDA transceivers according the manufacturers' specification.

These functions are called by the Framer or IrLAP processes, and developers have to provide the program codes for programming the IrDA transceivers according to manufacturers' specifications. The functions are stored in the PPSM-GT device driver directory, and developers have to use the same name syntax to effect the function call.

**Table 6.1 IrDA Transceiver Functions**

| Function name | Description |
|---|---|
| STATUS **IrdInitTransceiver**(void) | Called to initialize the specific transceiver used in the system. Called from the IRFRAMER_Init function (which is built into the stack). |
| STATUS **IrdShutDownTransceiver** (void) | If a given transceiver has a shutdown capability, this function forces it to shut down. This is ordinarily called by the upper layer application. |

**NOTE** The name function and the function prototype cannot be modified by the manufacturer.

7

# ISR Routines Services

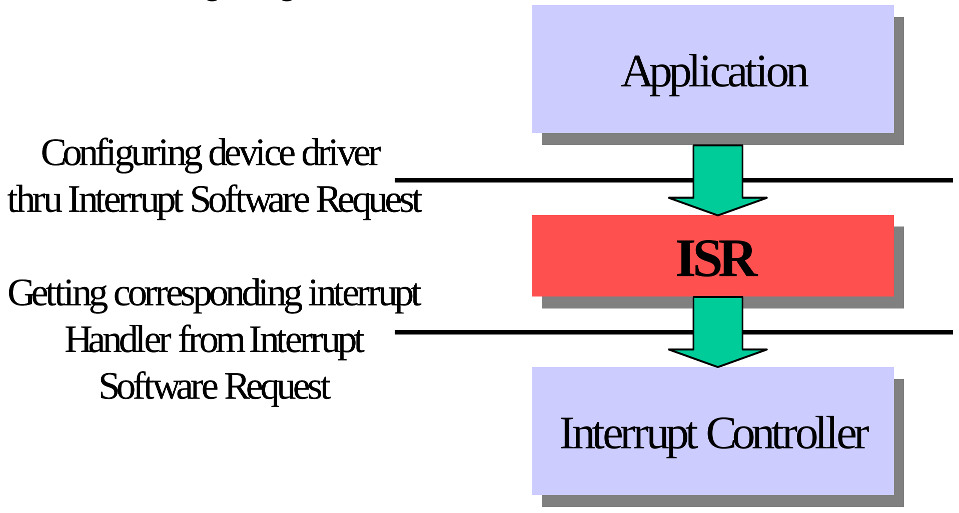
ISR Routines Services provides the interfaces for the User application to the DragonBall Interrupt Controller.

Interrupt Software Request is used as a device driver interface between the application and the hardware. It makes the drivers that are built on the top of DragonBall's interrupt controller configurable. Figure 7.1 shows the interface of ISR with application and the interrupt controller. The application gain access to the hardware through the ISR services. The application use the ISR service to configure the device driver, and the ISR services buffered the application form directly servicing the real time interrupt.

This chapter is presented into the following broad topics:

- ISR Services Fundamentals
- Programming using the ISR services
- Code Example

**Figure 7.1 ISR interfacing diagram**


Application
Configuring device driver thru Interrupt Software Request
ISR
Getting corresponding interrupt Handler from Interrupt Software Request
Interrupt Controller

## ISR Services Fundamentals

The PPSM-GT ISR services are hardware specific routines. The current version of the ISR services works only with Motorola M68VZ328 microcontroller. When using the PPSM-GT ISR routines service, the input modules required by the APIs can be found in Table 7.2 which shows the corresponding DragonBall-VZ interrupt flags against the modules. System integrator and software developer may request and assign handler to these interrupts through the IsrRequest( ) API, and when the interrupt services is no longer required, IsrRelease() API is used.

For each interrupt, only one handler can be assigned. The interrupt has to be released before being assigned to another handler.

**Table 7.1 DragonBall Interrupt Module**

| Modules | ISR_Module_FLAG |
|---|---|
| Emulator interrupt | ISR_EMIQ_FLAG |
| Sampling timer | ISR_SAM_FLAG |
| SPI1 | ISR_SPI1_FLAG |
| IRQ5 | ISR_IRQ5_FLAG |
| IRQ6 | ISR_IRQ6_FLAG |
| IRQ3 | ISR_IRQ3_FLAG |
| IRQ2 | ISR_IRQ2_FLAG |
| IRQ1 | ISR_IRQ1_FLAG |
| PWM2 | ISR_PWM2_FLAG |
| UART2 | ISR_UART2_FLAG |
| INT3 | ISR_INT3_FLAG |
| INT2 | ISR_INT2_FLAG |
| INT1 | ISR_INT1_FLAG |
| INT0 | ISR_INT0_FLAG |
| PWM1 | ISR_PWM1_FLAG |
| Keyboard | ISR_KB_FLAG |
| Timer2 | ISR_TMR2_FLAG |
| Real time clock | ISR_RTC_FLAG |
| Watch dog timer | ISR_WDT_FLAG |
| UART1 | ISR_UART1_FLAG |
| Timer1 | ISR_TMR1_FLAG |
| SPI2 | ISR_SPI2_FLAG |

## Configurable DragonBall-VZ modules

In DragonBall-VZ, some of the ISR modules have additional features that allow it's interrupt levels to be configurable from level 6 to level 1. Table 7.1 shows the list of DragonBall modules whose interrupt levels could be configured using the IsrSetIrptLv() API.

IsrGetIrptLv() and IsrIsIrptLv() are used for getting the modules interrupt levels and checking whether the module's interrupt is configurable respectively.

**Table 7.2 Table showing DragonBall-VZ configurable modules**

| **interrupt configurable modules** | **ISR_Module_FLAG** |
|---|---|
| SPI1 | ISR_SPI1_FLAG |
| UART2 | ISR_UART2_FLAG |
| PWM2 | ISR_PWM2_FLAG |
| Timer2 | ISR_TMR2_FLAG |

## Programming using the ISR services

Programming the ISR services, begin with requesting the interrupt service routine with IsrRequest(). For example, the following are the recommended steps to request for a UART interrupt if you are expecting one.

1. Plan and design your interrupt handler. That is how you are going to handler the UART interrupt when it occurred.
2. Use IsrIsInUse() to check whether the it is in use.
3. If not in use, then use IsrRequest() to request for the interrupt else wait until interrupt is available.
4. When using IsrRequest(), you need to provide type of interrupt requested by the module flag provided in Table 7.1 and the pointer to interrupt handler. Example 7.1 show how to use IsrRequest for UART interrupt.
5. When done with interrupt, use IsrRelease() to release interrupt.

## Requesting the ISR

| | |
|---|---|
| STATUS **IsrRequest**(U32 module, PFIRTHANDLER pfIrptHandler, U32 arg) | It is the function to request, configure and set the handler of an interrupt. The following are the input parameters described :<br>• module - ISR_Module_FLAG as shown in Table 7.1<br>• pflrptHandler - Point to ISR handler routine.<br>• arg - argument for ISR handler if any |

## Releasing the ISR

| | |
|---|---|
| STATUS **IsrRelease**(U32 module) | It is the function to release an interrupt. |

## Getting the Current Interrupt level

| | |
|---|---|
| U8 **IsrGetIrptLv**(U32 module) | It returns the current interrupt level of an interrupt. "0" means interrupt level is configurable. "0xFF" means interrupt is not available. |

## Checking the module interrupt level

| | |
|---|---|
| U8 **IsrIsInUse**(U32 module) | It tests if a module is in use. |

## Setting the Interrupt level

| | |
|---|---|
| STATUS **IsrSetIrptLv**(U32 module, U8 irptLevel) | It is configures the interrupt level of a module if it is configurable. |

## Code Example

**Listing 7.1 Example of requesting an interrupt**

```
VOID UartHandler(U32 arg)

/* Argument "arg" is not used, but it must be passed in */
/* Request UART interrupt, with a given handler and "0"  */
/* as argument "4" is ignored as the requested interrupt is  */
/* not configurable                                      */

    if IsrIsInUse(ISR_UART_FLAG)
    IsrRequest(ISR_UART_FLAG, (PFIRTHANDLER)UartHandler, 0);

/* Request UART interrupt, without a handler "4" is ignored */
/* as the requested interrupt is not configurable      */

    IsrRequest(ISR_UART_FLAG, NULL, 0);
```

**Listing 7.2 Example of releasing an interrupt**

```
/* Release UART interrupt                      */

    IsrRelease(ISR_UART_FLAG);
```

# Developing with System Services

The chapters in this section help answer the question, “How do you use the System Services to create a system?” System Services are essential services. Every PPSM-GT system need to use these services to ensure proper operation. System Services set up the backbone for the system and ensure that the system operates properly.

In this section, each chapter has three parts.

- The first part of each chapter discusses System Services fundamentals. It introduces the concepts that must be understood before using the services.
- The second part of each chapter explains the APIs—the interfaces and functions of each API. For details on each API, please refer to the PPSM-GT API reference document.
- The last part of each chapter consists of a short summary and a code example that shows how to use the APIs. Please note that the example mainly shows how to use a particular API and is not designed to address any specific problem. The examples should be taken as a reference and should not be copied blindly.

The following are the chapters in this section:

- Chapter 8, “Kernel Services”—introduces the PPSM-GT Kernel Services and explains how to use them to create tasks and semaphores as well as get system information.
- Chapter 9, “Memory Management Services”—introduces fundamental concepts regarding the PPSM-GT memory

structure and explains how to malloc and realloc memory and regions.

- Chapter 10, “Power Management Services”—introduces PPSM-GT power management and how to program to the system to use idle, doze, and sleep modes. This chapter also provides APIs for controlling I/O when entering and exiting the power modes.
- Chapter 11, “System Application Services”—introduces PPSM-GT Application concepts and explains how to use applications in your system.
- Chapter 12, “System Event Management Services”—introduces the system events and explains how to use system events for intertask communication.
- Chapter 13, “Software Timer Handling Services”—introduces the software timer and how to program and set software timers.

# 8 Kernel Services

The PPSM-GT Executive Kernel is a 32-bit multitasking kernel that encompasses many components. This chapter is devoted to the basic task services: creating a new task, starting an existing task, terminating the requesting task, changing the priority of a given task, and deleting the requesting task. There is also information on semaphore and how semaphore could be used to control the accessing of some shared resource. A code portion in which a task is accessing such a resource is often called a "critical region."

This chapter is organized into the following sections:

## Kernel Services Fundamentals

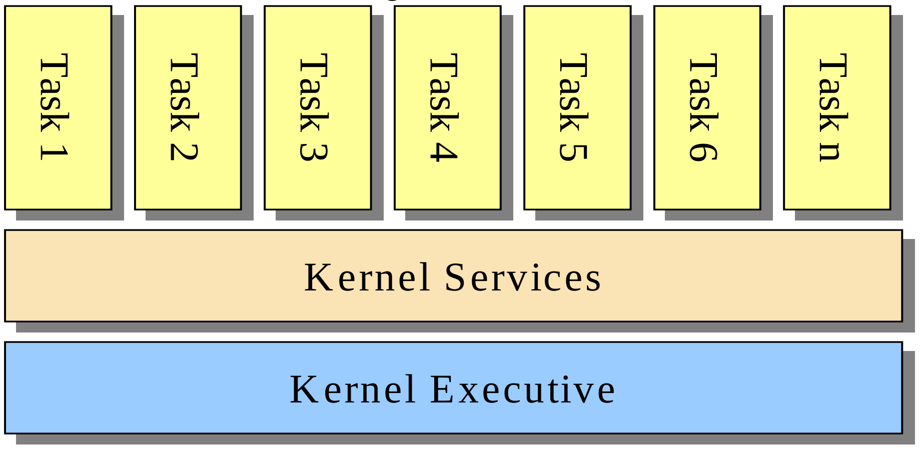
This section discusses broad concepts related to kernel services. The kernel executive is like the system control center. It controls the execution of tasks based on task readiness and priorities. For tasks that are ready to run, tasks with higher priorities are executed first.

Figure 8.1 shows the relationship between tasks, kernel services and the kernel Executive. Tasks have no direct access to the kernel executive. All tasks work through kernel services to access the routines in the kernel executive, which are responsible for task manipulation and kernel operation. The kernel services could be broadly classified as APIs for the following:

- Task Manipulation

- Semaphores
- Special Functions

**Figure 8.1 Kernel Block Diagram**



## Task Manipulation

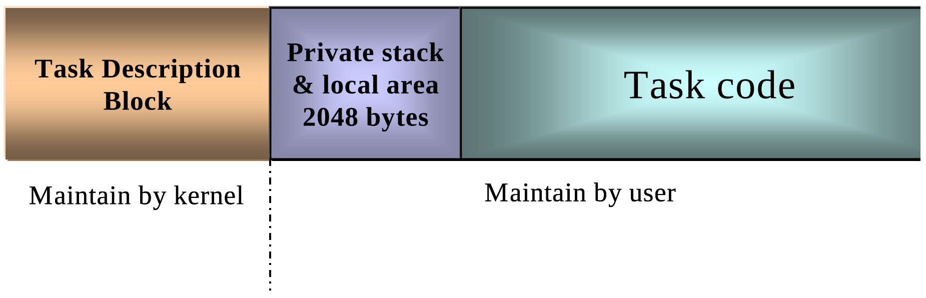
Tasks are essentially the basic units of execution for a real-time application. Tasks must be created and maintained by the kernel. Understanding the function of a task and its operation is useful when designing the applications for the system.

Tasks are essentially a series of separate component programs that can execute concurrently. Each task is a complete program that is capable of independent execution. Each task has a segment of code that it executes. Each task has its own private stack and its own local data areas. There are dedicated memory segments in which the task can keep procedure call parameters, return addresses, temporary data, and similar variables that are not shared with other tasks.

In theory PPSM-GT sets no restriction on the number of tasks it can support. However, in practice the number of tasks is determined by the amount of system memory available.

As shown in Figure 8.2, all tasks have three components: task code, a task stack, and a task description block.

**Figure 8.2 Task structure**



| Task Part | Description |
|---|---|
| Task code | The task code is the set of processor instructions that will be executed when the task executes. |
| Task stack | The task stack is a region of RAM that is reserved for the exclusive use of a single task; by default it is 2048 bytes. |
| Task description block | The task description block, or TDB, is a RAM data structure maintained by the kernel executive for each task created. It contains information on the task: its identity, priority level, and current active graphics context as well as the current state of the task. |

### Type of tasks

PPSM-GT supports two types of tasks: real-time tasks and time-sliced tasks. Real-time tasks, sometimes referred to as prioritized tasks, are executed as long as they have resources they need. Time-sliced tasks, sometimes referred to as background tasks, are executed only for a limited time—"quantum"—and then another time-sliced task may run.

Both types of tasks have priority levels that are set upon creation. There are 16 priority levels that PPSM-GT supports. The highest level is 15 and the lowest is level 0. The lowest level tasks are reserved by the kernel executive. A system integrator can only create tasks with priority levels from 15 to 1.

**Figure 8.3 PPSM-GT Task organization**

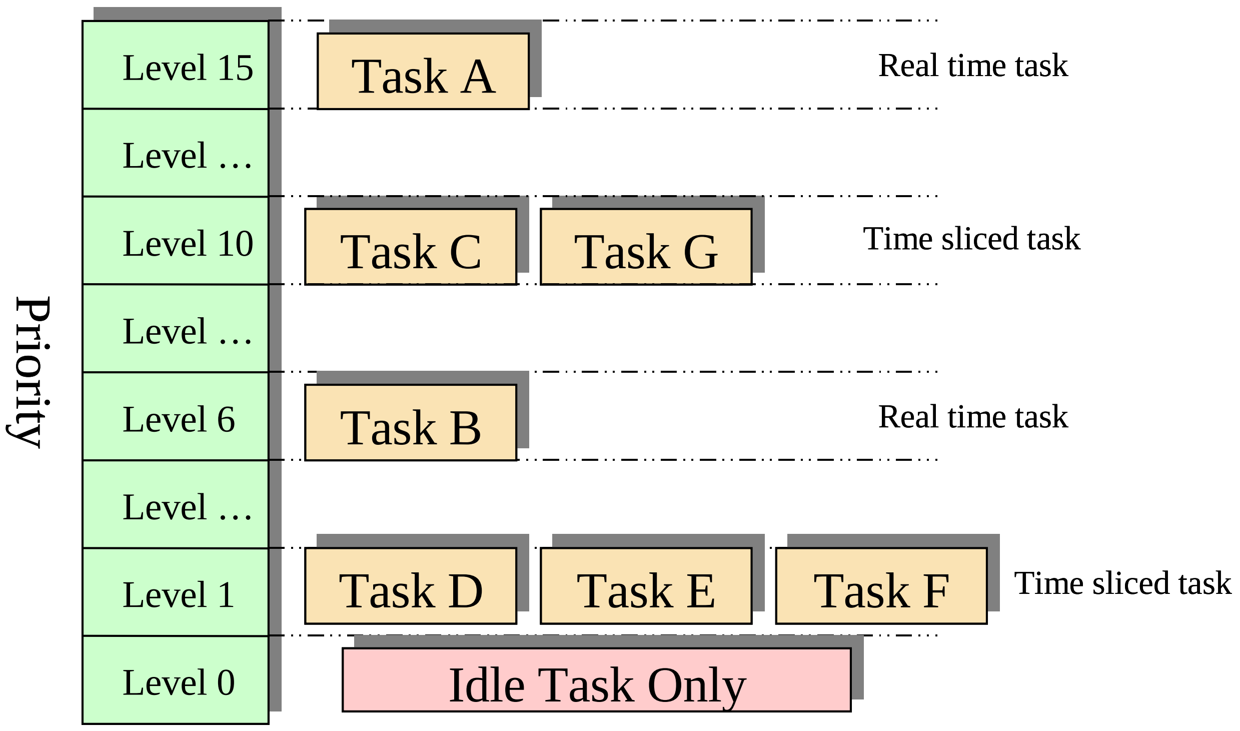


Figure 8.3 shows an example of how tasks are organized in PPSM-GT. In Figure 8.3 task C and task G form one group of time-sliced tasks at priority level 10, and task D, task E and task F form another group of time-sliced task at level 1. Task A and task B are real-time tasks. Priority level 0 is reserved for system idle tasks only.

The task priority level in PPSM-GT not only determines the execution rights of the task; it also determines the type of task. In PPSM-GT, if two or more tasks have the same priority, they will be classified as time-sliced tasks.

Both real-time and time-sliced tasks execute until one of four possible things happens:

1. The task completes its function and is self suspended or terminated.
2. The task is waiting for an event to occur.
3. The task is suspended.
4. The task is interrupted by some higher priority task.

In addition, time-sliced tasks execute until the quantum expires.

It is beneficial to note that time slices are not precisely timed and may be suspended or interrupted by real-time activities at any point of time. Time slices are also perturbed by sliced tasks blocking themselves. Therefore, time slicing should only be used when timing is unimportant.

### *The Idle task*

The idle task is a predefined system task in PPSM-GT. It has the lowest priority in the system; there should be no other tasks that can have a priority lower than or equal to the priority of the idle task.

The idle task will only run when all the other tasks are not ready. PPSM-GT uses the idle task to perform some memory clean up and power mode maintenance.

The idle task cannot be terminated or suspended by other tasks.

## Task Transaction

Figure 8.4 shows the state transaction diagram of a task. Depending on the nature of the system design, a task could change from one state to another until it is finally terminated. Table 8.1 shows the definition of each state of the task.

**Figure 8.4 Task transaction state diagram**

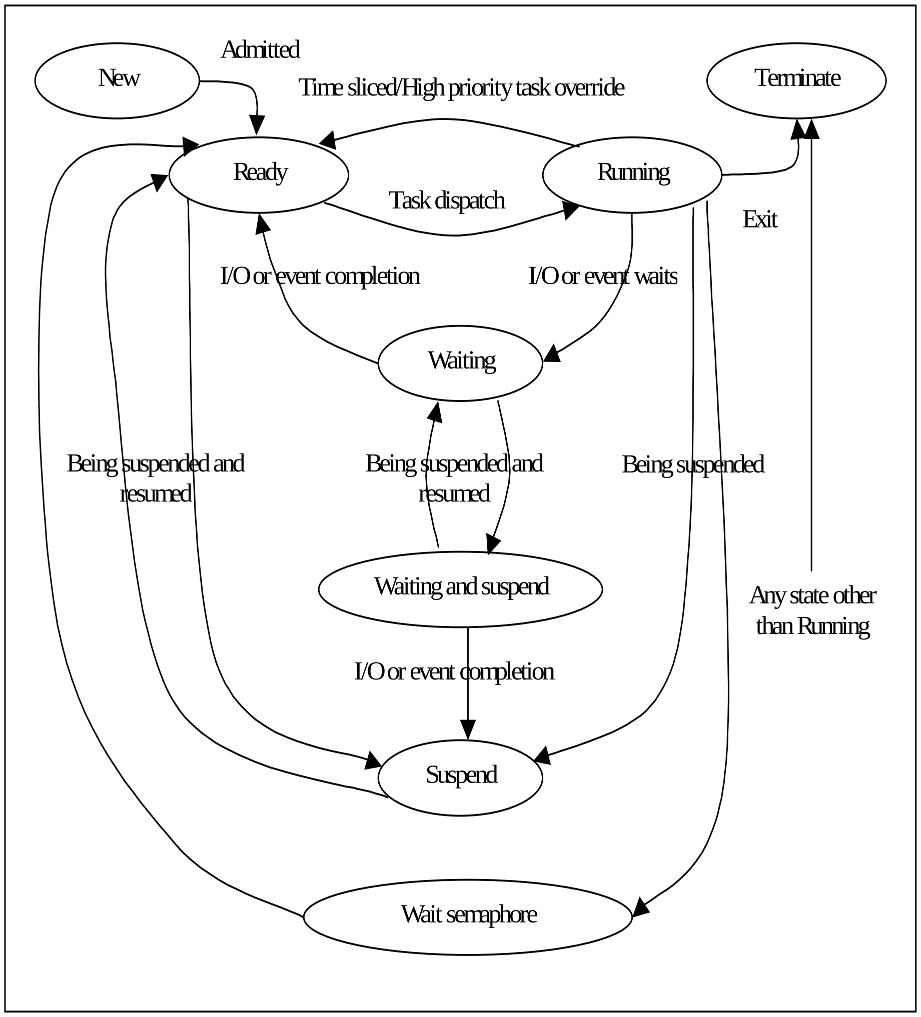


**Table 8.1 Meaning of each state**

| State | Description |
|---|---|
| New | A task is created and not ready to run. |
| Ready | A task is ready to run. |
| Running | A task is running. There is always only one in a system. |
| Terminate | A task is terminated. |
| Waiting | A task is waiting for some I/O or events. |

| State | Description |
|---|---|
| Suspend | A task is suspended. |
| Waiting and suspend | A task is suspended and also waiting for some I/O or events. |

### Task Status

Every task has a task state. It can be NEW, TERMINATE, READY, WAIT, RUN, SUSPEND, WAIT_AND_SUSPEND or WAIT_SEMAPHORE. There is always only one running task in the system. It is the highest priority ready-to-run task.

The WAIT status indicates that a task is waiting for an event (such as I/O) to happen. SUSPEND status indicates that a user explicitly suspended a task. The user must explicitly resume the operation of this task.

### Multiple level of suspension

A task can be multi-suspended up to 7 levels. This is a cumulative action such that if a task is suspended 5 times, it has to be resumed 5 times before it is active.

## Semaphores

In most applications, tasks must share sets of data, such as a table that is read by one task and updated by another. A second example of shared data consists of the global variables within a non-reentrant procedure that could be called by different tasks.

A segment of code in which a task is accessing some shared resource is often called a "critical region" with respect to that resource. Not every reference to shared data forms a critical region. A region is critical only if there could be harmful interaction because of the sharing of the resource. For data, this means that variables are both shared and alterable.

Shared resources must be protected against potentially harmful interactions by permitting only one task at a time to enter a critical region. In the example of sharing a common table, while the data is being read from the table, other tasks that update the table should be blocked; while a task is updating the table, other tasks that need to read data from the table should be blocked.

Critical regions must be protected by guaranteeing one-task-at-a-time access. PPSM-GT uses a counting semaphore design to handle the accessing of the critical region.

Kernel services enable a task to create, delete, signal and wait on all semaphores. This allows a task to protect the access of the critical region when required and to open up the access of the critical region when protection is no longer required.

The basic steps for using the semaphore are as follows:

1. Create the semaphore with KnlCreateSemaphore().
2. Set the maximum and initial number of tasks that could use the semaphore with KnlSetSemaphore().
3. Use the semaphore with KnlWaitSemaphore().
4. Release the semaphore when done with KnlSignalSemaphore().
5. Delete the semaphore when it is not required with KnlDelSemaphore().

### Waiting for Semaphore

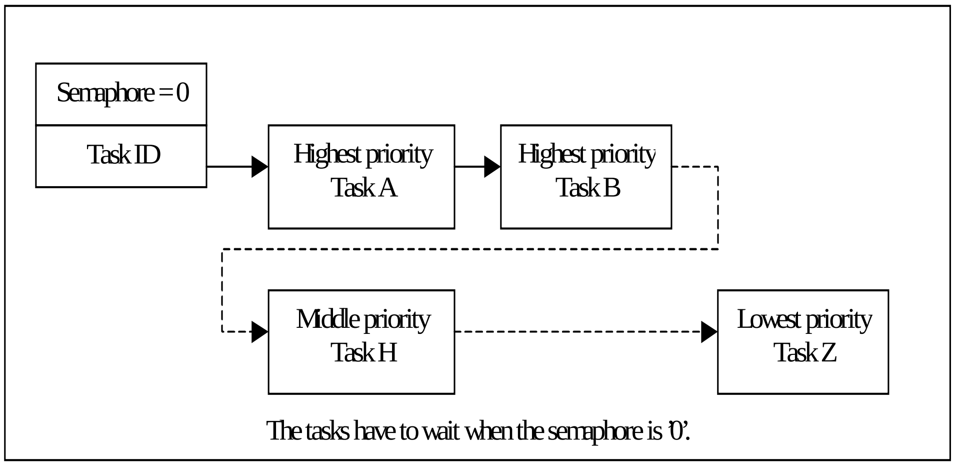
There are a few options while waiting for a semaphore. There is a choice of queue sequence and wait duration. The wait duration varies from “wait forever” to “no waiting.” When the no waiting option is selected, the system checks whether the semaphore is available. If it is not, the system continues executing the next command.

The two queue options are described as follows.

#### *Priority queue*

For this queuing option, tasks wait for a semaphore in a queue based on their priority levels. That is, if task A has a priority of 5 and task B has a priority of 2, when a semaphore is available, the kernel signals task A first. Figure 8.5 show the priority queue sequence.

**Figure 8.5 Tasks queue up to wait for SIGNAL based on priority queue**



### *FIFO queue*

For this queuing option, tasks wait for a semaphore in a queue based on FIFO (first in first out). That is, if task A is waiting for a semaphore earlier than task B is, when a semaphore is available, the kernel signals task A first.

# Programming using Kernel services

## Task Manipulation Services

### Creating a Task

| Function | Description |
|---|---|
| STATUS **KnlCreateTask**(P_TASK_ID pTaskId, P_VOID pFunc, const TEXT pName[], U32 stackSize, S8 priority, KNL_MODE mode) | It creates a task without arguments from a given function. |
| STATUS **KnlCreateTaskWith**(P_TASK_ID pTaskId, P_VOID pFunc, const TEXT pName[], U32 stackSize, U32 arg, S8 priority, KNL_MODE mode) | It creates a task without arguments from a given function and allows user to pass an unsigned integer parameter into the task on creation. |

Both APIs create a task from a function at a memory location "Func" with a stack size of "stackSize" and an assigned priority of "priority." If a zero "stackSize" is given, the default value of 2048 bytes is used. This is the minimum required memory for a task without any local variables and sub-routine calls.

The mode parameter should be set at the system default value of "1."

KnlCreateTaskWith( ) has an extra parameter that allows users to pass an unsigned integer parameter into the task on creation.

In the main ( ) routine, only the highest priority task—priority level 15—can be created. The system does not allow any lowest priority task (priority level 0) to be created. The lowest priority task is solely occupied by the idle task.

In normal operation, a task cannot create another task with a higher priority. That is, if a task has a level 4 priority, it cannot create any task with a priority level higher than priority level 4.

KnlCreateTask() and KnlCreateTaskWith( ) are not suitable to be used in interrupt handling routines.

PPSM-GT supports dynamic task creation. System integrators and software developers are free to create real-time tasks and time-sliced tasks. PPSM-GT does not impose any restrictions on time-sliced tasks. The flexibility and creativity is extended to developers.

**WARNING!** In PPSM-GT, the only difference between a real-time prioritized task and a time-sliced task is the priority. In other words, when two or more tasks have the same priority level, they become time-sliced tasks. System integrators and software developers are therefore strongly advised to practice good programming control on the creation of task. A time critical real-time task could become a time-sliced task if there is another task with the same priority level.

### Getting and Changing Task Priority

| | |
|---|---|
| STATUS **KnlGetPriority**(TASK_ID taskId) | Return the target task priority. |
| STATUS **KnlChangePriority**(TASK_ID taskId, U8 priority) | Change the priority of the target task based on the specified task ID. If the task ID is zero, the system will treat it as the current task and return the current task priority when KnlGetPriority() is called. If KnlChangePriority() is called with taskId = KNL_CURR_TASK, then the current task priority will be changed. |

The valid range of priority levels is 0 to 15; level 0 is the lowest permissible level and level 15 is the highest permissible level. The changed priority level cannot be higher than the calling task's level. That is, if task A has a priority level of 4 and task A wants to change the priority level of B, the maximum level to which task B can be changed is level 4.

Take note also of the Warning in the preceding section.

KnlChangePriority() cannot be used in interrupt handling routines. The system will return an ERR_KNL_IN_IRPT error message if this API is called from an interrupt routine.

### Getting Task ID and Deleting task

| | |
|---|---|
| TASK_ID **KnlGetTaskId**(VOID) | Get the current running task ID. |
| STATUS **KnlDeleteTask**(TASK_ID taskId) | Terminate a running task according to the taskId provided. A task cannot use KnlDeleteTask() to terminate itself. |

KnlDeleteTask is not suitable to be used in interrupt handling routines.

WARNING! Developers are strongly discouraged from using the KnlDeleteTask API to terminate a task; sources that are created by the task may not be freed properly.

### Setting Task Event Port for receiving events

| | |
|---|---|
| EVTPORT_ID **KnlGetEventPort**(TASK_ID taskId) | KnlGetEventPort(), when called, returns a positive EVTPORT_ID value if the task has an event port. |
| STATUS **KnlSetEventPort**(TASK_ID taskId, EVTPORT_ID portId) | KnlSetEvenPort() sets up the task with the event port. |

An event port is a mechanism that enables a task to receive an event. By default, when a task is created, the event port is enabled to receive an event. KnlSetEventPort( ) allows an event port to be enabled or disabled by choice.

### Suspending a Task

| | |
|---|---|
| STATUS **KnlSuspend**(TASK_ID taskId) | When a task is created, it is always in the new state. It will change from the new state to the ready state when it is scheduled to run. |

KnlSuspend() suspends the task specified by taskId.

KnlSuspend() is not suitable to be used in interrupt handling routine and has no effect on the idle task.

### Self Suspending task for time interval

| | |
|---|---|
| STATUS **KnlSuspendFor**(TICK milliseconds, SWT_ID swtId) | This API allows the calling task to be self-suspended for the given number of milliseconds specified by the TICK.<br>The minimum time interval is 1 millisecond and the maximum is 24 hours as defined in the software timer.<br><br>The self-suspended task will be reactivated automatically once the time-out interval elapses.<br><br>This API does not create a software timer. The software timer has to be created before calling this API, and the system will not delete the software timer after used. User need to ensure that the software timer is deleted if not used.<br><br>This API also cannot be used if task swapping is disabled. |

### Resuming a Suspended Task

| | |
|---|---|
| STATUS **KnlResume**(TASK_ID taskId) | To resume a suspended task, KnlResume() can be used. When called, it resumes the task specified by taskId. |

## Disabling Task Swapping

| | |
|---|---|
| STATUS **KnlDisableSwap**(VOID) | To disable task swapping. Once activated, no task swapping is possible regardless of the task priorities, including swapping to the idle task for power management. In this situation, the system becomes a single task operation. Therefore developers must observe the following cautions when using this instruction:<br>• Enable task swapping when task swapping disabling is no longer required to allow the system to perform power management.<br>• Ensure that the task that activates the KnlDisableSwap does not have an infinite loop that hangs up the system. The system will not be able to recover in such a situation except through a hard reset.<br>• The task cannot self terminate. The system will not allow the task to self terminate.<br>• For a time-sliced task, the task will continue to operate outside the quantum when it expires. The time-sliced task will only be suspended when task swapping is enabled. Yielding of the quantum is also not allowed when task swapping is disabled. |

### Enabling Task Swapping

| | |
|---|---|
| STATUS **KnlEnableSwap**(VOID) | To enable task swapping. Once activated, task swapping is enabled, and task execution will be based on the next highest priority task that is ready to run. |

This API must not be called in main().

### Yielding the execution quantum for Time-slicing task

| | |
|---|---|
| STATUS **KnlYield**(VOID) | In time-slicing task operation, a task can yield its operation time (quantum) when it has finished it function. KnlYield() is a special case of self suspension and is applicable for time-sliced tasks only. When called, a task voluntarily gives up the remaining quantum in its time slice. Once the quantum is yielded, it cannot be regained. |

## Semaphore Services

### Creating Semaphore

| | |
|---|---|
| SEMA_ID **KnlCreateSemaphore(**VOID) | KnlCreateSemaphore( ) creates a semaphore structure and returns the semaId. |

## Deleting Semaphore

| | |
|---|---|
| STATUS **KnlDelSemaphore**(SEMA_ID semaId, U8 flag) | KnlDelSemaphore() deletes and frees up the semaphore structure. If the semaphore is in use by any task (that is, if any task is waiting for it), it cannot be deleted or freed. The U8 flag is currently not implemented and should be set to “0” when using KnlDelSemaphore( ). |

## Setting the Semaphore

| | |
|---|---|
| STATUS **KnlSetSemaphore**(SEMA_ID semaId, U16 max, U16 init, BOOL fifo) | When a semaphore is created, max and init are zero; that is, no task could use it. If any task calls KnlwaitSemaphore() when the max value = 0, the task will hang up waiting for a semaphore that is not set. |

The maximum and initial value of the semaphore could be set by KnlSetSemaphore( ). When called, it sets the max and init values. The max value will determine the number of users that are allowed to use the semaphore. The init value is the initial value of the semaphore usage. It could be any number smaller or equal to the max value. Normally it is 0.

The BOOL option is for setting the queue type. Choose between FIFO or priority queue. It is a boolean input: TRUE for fifo queue and FALSE for priority queue. When priority queue is selected, the waiting for a semaphore is based on task priority. For FIFO queue, the waiting for a semaphore is based on first in first out.

KnlSetSemaphore() could be used also at other times to change the max and init values for an existing semaphore. However, it cannot be used on a semaphore when other tasks are waiting on the semaphore.

### Signaling the Semaphore

| | |
|---|---|
| STATUS **KnlSignalSemaphore**(SEMA_ID semaId) | KnlSignalSemaphore(), when called, will increase the init semaphore value by 1 up to or equal to the max value. Under normal operating conditions, tasks should call KnlSignalSemaphore() when they are done with the semaphore to release the semaphore to other tasks. |

KnlSignalSemaphore() could also be used to increase the number of tasks using the semaphore up to the max value. Increasing the init value by 1 would increase the number of tasks able to use the semaphore by 1.

KnlSignalSemaphore() is not suitable to be used in interrupt handling routine

### Waiting for Semaphore

| | |
|---|---|
| STATUS **KnlWaitSemaphore**(SEMA_ID semaId, TICK milliseconds) | When KnlWaitSemaphore() is called, it checks the semaphore init value. If it is larger than 0, the value is decreased by 1. The task then could use the semaphore. |

If the init value of the semaphore is equal to 0, the calling task is blocked if the input TICK is not "0," and its taskId is put into a priority and first-in-first-out queue in the semaphore queue. This is a form of self suspension mode.

The TICK input is a time interval input in milliseconds that informs the system how long the calling is willing to wait for the semaphore. The following are the options:

- TICK equals "0": the waiting interval is zero; there is no waiting. Under this condition, the system will check for the semaphore. If it is unavailable, the system will continue executing the next instruction without waiting for the semaphore. If it is available, the semaphore is deducted by 1.

- TICK equals any value between 1 and the max value defined for TICK in the software timer (by default it is 24 hours): the system will wait for the semaphore for a number of milliseconds equal to that value. The task will only exit the waiting loop when the semaphore is available or the wait semaphore timer expires.
- TICK equals SEMA_WAIT_FOREVER: the system will wait forever until the requested semaphore is available. For the semaphore to be available, its init value must be greater than zero.

The semaphore-wait operation is not allowed in any interrupt, since it may block any operation that is after a semaphore-wait operation in an interrupt. Therefore, any function that calls semaphore-wait also cannot be called within an interrupt.

### Checking for Semaphore ID

| | |
|---|---|
| U8 **KnlIsSemaId**(SEMA_ID semaId) | KnlIsSemaId() is for checking whether an ID is a semaphore ID. When called, it returns the boolean value "TRUE" or "FALSE." |

## Special Functions

PPSM-GT also provides many other APIs. Some are for status checking, and others are for setting the graphic context for a display property. For details of the graphic context, refer to the graphic chapter.

### Getting Graphic context

| | |
|---|---|
| GC_ID **KnlGetGC**(TASK_ID taskId) | It returns the graphic context, gcId, for the task. If taskId is zero, it returns the graphic context of the current task. |

### Binding Graphic Context to Task

| | |
|---|---|
| STATUS **KnlBindGC**(TASK_ID taskId, GC_ID gcId) | Graphic contexts, GCs, are drawing properties for PPSM-GT graphic services. They need to bind to a task so that the task can use the graphic services to create graphic. KnlBindGC() binds the task with the graphic context.<br>If taskId is KNL_CURR_TASK, then the GC will be bound to the calling task. |

### Getting Task Memory usage.

| | |
|---|---|
| STATUS **KnlGetMemUsed**(TASK_ID taskId) | It returns the memory used by the task. |

### Getting the OS Version

| | |
|---|---|
| STATUS **KnlGetOSVersion**(P_U32 major, P_U32 minor) | It returns the current PPSM-GT version. |

### Getting the Stack Info

| | |
|---|---|
| STATUS **KnlGetStackInfo**(TASK_ID taskId, P_VOID *start, P_VOID *end) | It returns whether the stack has been overflowed. To get the stack info for the current running task, use "KNL_CURR_TASK" in place of taskId. |

### Getting Kernel Status

| | |
|---|---|
| U32 **KnlGetStatus**(TASK_ID taskId) | It returns the status of the task (NEW, TERMINATE, READY, WAIT, RUN, SUSPEND or WAIT_AND_SUSPEND). |

### Checking whether the Kernel is currently serving an interrupt

| | |
|---|---|
| U8 **KnlIsInIrpt**(VOID) | It tests whether the kernel is currently serving an interrupt. The system will return a Boolean "TRUE" or "FALSE." |

## Summary

Kernel Services provide the basic building blocks for the system. Task manipulation services help to create a new task, start an existing task, terminate the requesting task, change the priority of a given task, and delete the requesting task.

Semaphore services enable semaphores to be used to control the access of some shared resource in a "critical region" of code. As required by the application, it could wait forever, wait for a period or not wait for a semaphore. This functionality is provided to cater to different needs based on the nature of the application.

Special functions are a group of APIs provided for checking the status and the information of the kernel. Most of the information provided is informative and useful especially during the debugging and development of the system.

## Code Examples

The following examples are provided to show the usage of the kernel services. They are not intended to solve any problem and should be used as a reference. They should not be used directly in an application without modification.

**Listing 8.1 Usage Task Manipulation in Kernel Services**

```
/* This example shows the usage of KnlTaskCreate() and
KnlTaskCreateWith() to create 3 tasks: Task A, B, & C. It also
provide a simple example to show the usage of KnlSuspendFor() to
self suspend task A for 200 milliseconds.*/


/* Variable declaration*/
  #defineNORMAL_PRIORITY 6
  #define DEFAULT_MODE 1

  TASK_IDTaskAId, TaskBId, TaskCId;
  const TEXTTaskNameA= {'T','A','S','K','_','A',0};
  const TEXTTaskNameB= {'T','A','S','K','_','B',0};
  const TEXTTaskNameC= {'T','A','S','K','_','C',0};
  SWT_ID swtId;
  STATUS status;

  P_U32 TaskCode[] = {
    (P_VOID)TaskA,
    (P_VOID)TaskB,
    (P_VOID)TaskC
    };
  U32 Argument, Priority, TaskId;
  EVTPORT_ID PortAId, PortBId;

void taskcreate()
{
  status = KnlCreateTask(&TaskAId, (P_VOID) TaskA, TaskNameA,
         4000, NORMAL_PRIORITY, DEFAULT_MODE);
  Argrument = 2200;
  status = KnlCreateTaskWith(&TaskBId, (P_VOID) TaskB, TaskNameB,
         4000, Argument, NORMAL_PRIORITY+1, DEFAULT_MODE);

  /* Creating a time slicing task */
  status = KnlCreateTask(&TaskCId, (P_VOID) TaskC, TaskNameC,
         4000, NORMAL_PRIORITY, DEFAULT_MODE);

  return;
}
void TaskA ()
```

```
{

/* Do anything for this task */
  ...

  /* Example of self suspending task A for 200 milliseconds*/

// create a SWT
  swtId = SwtCreate();
// suspend for 200 milliseconds
  KnlSuspendFor(200, swtId);

// perform other task A activities

  ..........

// delete the SWT when no longer needed

  SwtDelete(swtId);

  }

void TaskB ()
{
  /* Do anything for this task */
  ...
}

void TaskC
{
  /* Do anything for this task */
  ...
}
```

**Listing 8.2 Usage of Semaphore Services**

```
/* Variable declaration*/
  #defineNORMAL_PRIORITY 6
  #define DEFAULT_MODE 1

  TASK_IDTaskAId, TaskBId;
  const TEXTAppNameA= {’A’,’P’,’P’,’_’,’A’,0};
  const TEXTAppNameB= {’A’,’P’,’P’,’_’,’B’,0};
  STATUS status;
  U_8 done;

  P_U32 AppCode[] = {
    (P_VOID)AppA,
    (P_VOID)AppB
    };
  U32 Argument, Priority, TaskId;
  EVTPORT_ID PortAId, PortBId;
  SEMA_ID RedFlag;

void semaphore()
  {

  status = KnlCreateTask(&TaskAId, (P_VOID) AppA,AppNameA,
         4000, NORMAL_PRIORITY, DEFAULT_MODE);
/* Create and set the semaphore to have a max of 5 task waiting
for it, and the queue type is FIFO*/
  ReadFlag = KnlCreateSemaphore();
  status = KnlSetSemaphore(RedFlag, 5, 0, TRUE);
  }

viod AppA()
  {
/* wait for the semaphore until available*/
    status = KnlWaitSemaphore(RedFlag, WAIT_FOREVER);
      :
      :
      :
/* Signal semaphore when done*/
    KnlSignalSemaphore(RedFlag);

/* Check whether the Id belong is a semaphore Id*/
    done = KnlIsSemaId(RedFlag)


/* Delete the semaphore*/
    KnlDelSemaphore(RedFlag, 0)
```

# 9 Memory Management Services

Most devices use two types of memory: volatile and non-volatile memory. Volatile memory is memory that loses the stored data after powering off—for example, SRAM and DRAM. Non-volatile memory is memory that preserves the stored data after powering off—for example, FLASH. Normally, volatile memory is used for code execution and the storage of temporary information. Non-volatile memory is used for code and permanent data storage. PPSM-GT memory management tools handle only volatile memory, such as SRAM and DRAM. Developers need to handle data requiring non-volatile memory themselves.

This chapter has the following sections:

- Memory Management Fundamentals
- Programming using Memory Services
- Summary
- Code Examples

## Memory Management Fundamentals

In PPSM-GT, the entire memory is declared as a malloc pool during set up and can be divided into user-defined regions during execution. A region is defined as a block of continuous memory, and the minimum size of a region is 24 bytes.

The size of the malloc pool is the total physical memory available and needs to be specified in the PPSMspc.c file. Incorrect specification of the size of the malloc pool will result in errors in memory management.

PPSM-GT does not limit the number of regions created so long as the total memory of all regions added together is smaller than the size of the malloc pool. Memory region APIs are provided for adding, deleting, changing and resetting memory regions when desired.

By default, a system region is created by PPSM-GT. Usually, it is the largest memory region. The system uses the default region for creating tasks, graphic contexts, input contexts, and so on. It will also allocate memory from the default region when MemMalloc() and MemCalloc() APIs are used. The starting address and the size of the system region is declared in a linker specification file.

If no other regions are created, PPSM-GT will perform all memory-related activities in the system region. Exceptions are the MemMallocFrom() and MemCallocFrom() APIs, where the region specified in the APIs must be an existing region.

**Figure 9.1 Memory mapping of the system**

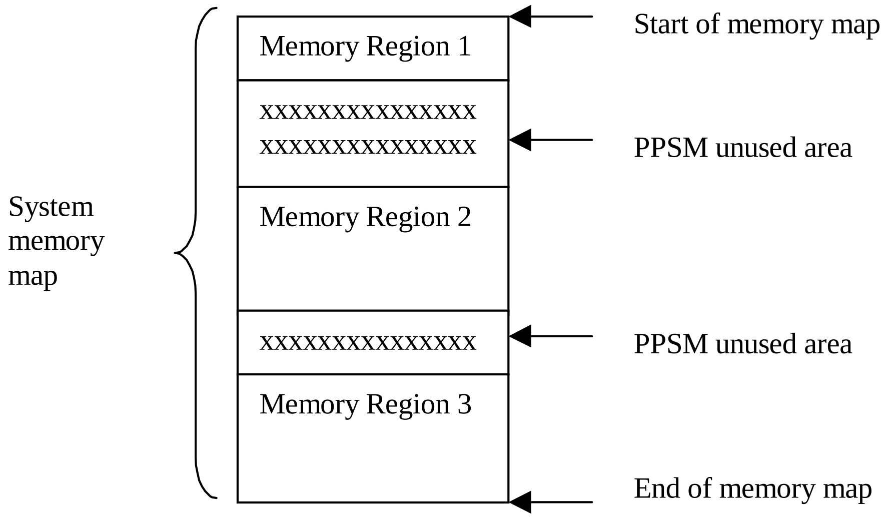


Figure 9.1 shows an example of how system memory could be mapped in PPSM-GT. Note that memory regions do not have to be continuous; there can be unused areas between them.

## Memory declaration

There are two files that store the information on memory declaration in PPSM-GT. Both of the files are device specific and need customization. The two files are:

1. Linker dependent specification file
2. PPSM-GT specification file (PPSMspc.c)

### Linker Dependent Specification File

The linker dependent specification file is normally used by the linker to allocate the specified item to the location declared in the file—for example, the location of the ROM and RAM. This file must be customized to the hardware, and depending on which development tools is used. For Metrowerks CodeWarrior user, the file is a .lcf file and for Single step SDS user, the file is a .spc file.

In the PPSM-GT environment, the linker dependent specification file is used to specify the starting address of the ROM and RAM only. The size of the ROM and RAM need not be specified here. They could be set to any value, including zero, as PPSM-GT does not derive the size of the ROM and RAM from those values. The PPSM-GT memory manager will control the memory allocation. The following is an example for memory definition in the linker dependent specification file.

```
MEMORY {
ROM (RX) : ORIGIN = 0x00001000, LENGTH =
0x00000000
RAM (RW) : ORIGIN = AFTER(ROM), LENGTH =
0x00000000
              }
```

The preceding example shows that the location of ROM is at 0x00001000. This location could be changed to any location required by the hardware. The RAM is defined as AFTER(ROM)—that is, the compiler will locate the RAM after the ROM is located. The length of both ROM and RAM are defined as 0x00000000. This does not mean that the size of ROM and RAM is zero. The value is actually “don’t care” because PPSM-GT does not read those values. For more detail, please refer to Metrowerks CodeWarrior document entitled Targeting_Embedded_68K.

### PPSM Link-time Specification File

This file specifies the boundary addresses for the system region. It defines the location of the system region and not the actual starting address of the system region, since the actual starting address is based on code size, constant, and string size declaration, which can only be determined by the compiler. The following code examples illustrate the differences in system starting addresses based on the code size, constant, and string declaration.

#### Example 1

```
In .lcf file
               MEMORY {
               ROM (RX): ORIGIN = 0x00000000, LENGTH = 0x00000000
               RAM (RW): ORIGIN = AFTER(ROM), LENGTH = 0x00000000
                                  }
               In PPSMspc.c
                    gMemMap={0, 0x1FFFFF}
```

The code size plus the constant and string declaration equals 0x1526.

Therefore, the starting address of the system region is 0x1526. The region from 00x0000 to 0x1525 is used by code size and constant and string declaration.

#### Example 2

```
In .lcf file
               MEMORY {
               ROM (RX) : ORIGIN = 0x00000000, LENGTH =
               0x00000000
               RAM (RW) : ORIGIN = AFTER(ROM), LENGTH =
               0x00000000
               }
               In PPSMspc.c
```

```
gMemMap={0x100000, 0x1FFFFF}
```

The code size plus the constant and string declaration equals 0x1526.

Therefore, the starting address of the system region is 0x100000. The code size and constant and string declaration total only 0x1526 and do not affect the starting address of the system region, which is declared at 0x100000.

The two examples show that proper definition of the system region is required for proper system operation. The system region is the largest memory region, and the system uses it as the default region for creating tasks, graphic contexts, input context, and so on. It will also allocate memory from the default region when MemMalloc() and MemCalloc() APIs are used.

In Example 1, the system does not start at 0x00000, as specified, due to the considerations for the location of the ROM and the code size plus constant plus string declaration.

## Actual available memory areas

The starting and ending addresses specified in the gMemMap are just the physical boundaries of the regions, and may not represent the actual physical memory available. For example, assume that the starting address is 100000 and the ending address is 1FFFFF for region A. This means that range A has been assigned 1 Mbyte of memory, but how much actual memory is in region A will depend on how much physical memory is available.

## Defining Memory Regions

Beside the system memory region, the rest of the memory regions are defined by the applications whenever required by the memory APIs. PPSM-GT provides APIs for adding, deleting, resetting and changing the memory regions. Table 9.1 presents five recommended steps for defining memory in PPSM-GT.

**Table 9.1 Basic steps for memory declaration**

| Step | Description |
|---|---|
| Step 1 | Determine how much memory is required by the system. Example: 4M of RAM. |
| Step 2 | Set the amount of RAM available as the system malloc pool. |
| Step 3 | Edit the PPSMspc.c file to specify and declare the memory system region. |
| Step 4 | Add additional regions in the application when required. |
| Step 5 | Delete regions that are no longer used. |

## Programming using Memory Services

### Allocating Memory

| | |
|---|---|
| P_VOID **MemMalloc**( U32 size) | Allocates memory in a default region to the calling task |
| P_VOID **MemMallocFrom**( MEM_REGION_ID regionId, U32 size, TASK_ID taskId) | Allocates memory in a specified region to the target task |
| P_VOID **MemCalloc**( U32 size) | Allocates memory in a default region to the calling task and initializes memory to zero. |
| P_VOID **MemCallocFrom**(MEM_REGION_ID regionId, U32 size, TASK_ID taskId) | Allocates memory for the specified region to the target task and initializes memory to zero |

Memory can be allocated to the application at run time. PPSM-GT 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 MemMalloc( ) and MemMallocFrom( ) are called. The difference between MemMalloc( ) and MemMallocFrom( ) is that

MemMallocFrom( ) allows memory to be allocated from a defined region and to a task specified by the taskId.

MemCallocFrom( ) performs the same function as MemMallocFrom( ) and initializes to zero when used.

No automatic boundary checking is performed on the memory when used by the caller.

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

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

These APIs are not recommended to be used inside interrupt handling routines.

## Freeing Memory

| | |
|---|---|
| void **MemFree**( P_VOID pUsedMem ) | MemFree( ) is used to free the area pointed to by "pUsedMem," which was previously allocated by MemMalloc( ), MemCalloc( ), MemMallocFrom( ), MemCallocFrom( ) or MemRealloc( ). |

When an application finishes with a block of dynamically allocated memory, the memory can be recycled by using MemFree( ). It puts the memory block back into the system heap, and the memory is ready for allocation again. MemFree( ) combines the freed area with any adjacent free areas at the same time.

The pointer passed into this routine must be a valid pointer returned from MemMalloc( ), MemCalloc( ), MemMallocFrom( ), MemCallocFrom( ) or MemRealloc( ).

These APIs are not recommended to be used in interrupt handling routines.

## Reallocating Memory

| | |
|---|---|
| P_VOID **MemRealloc**( P_VOID pOld, U32 size ) | PPSM-GT supports dynamic memory reallocation in the event of new requirements that make the current memory allocation insufficient. Calling MemRealloc( ) will reallocate the area of memory pointed to by "pOld," changing its size to "size" bytes. |

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

If the current memory can accommodate the increase in size of "size" bytes, then the Old address will be maintained. Otherwise the system will look for a new block of memory with a size of "size" bytes for the new request.

The system returns the address of a memory area containing the same initial bytes as the old memory area up to the smaller of the old and new sizes. If the returned address is different from "pOld," the old memory area is freed.

If the request cannot be satisfied, NULL is returned, and the old memory area remains allocated and its contents remain the same.

`MemRealloc( 0,  size )` acts like `MemMalloc( size )`.

`MemRealloc( pOld, 0 )` does `MemFree( pOld )` and returns NULL.

## Copying Memory

| | |
|---|---|
| STATUS **MemCopy**(P_U8 srcPtr, P_U8 destPtr, U32 size) | MemCopy( ) is for memory copying from one region to another. This tool can cope with overlapping areas. It performs memory copying in 32-bit operations whenever possible. |

## Inquiring Memory

| | |
|---|---|
| STATUS **MemGetAvailSize**(MEM_REGION_ID regionId, P_U32 pSizeAvail) | When MemGetAvailSize( ) is called, PPSM-GT returns the number of bytes of memory on the system that can be allocated through MemMalloc( ), MemMallocFrom( ), MemCalloc( ), MemCallocFrom( ) or MemRealloc( ). |
| S32 **MemGetAvailStack**(void) | When MemGetAvailStack( ) is called, PPSM-GT returns the total number of bytes of the stack that can still be used by the current task. A positive returned value indicates that the stack has not been used up; a negative value implies that the stack has already overflowed. |
| STATUS **MemGetOrgRegionSize**(MEM_REGION_ID regionId, P_U32 pSize) | MemGetOrgRegionSize( ) returns the original number of bytes of memory in a region before the region’s allocation to an application. |
| `STATUS` **`MemGetLargestBlk`**`(MEM_REGION_ID regionId, P_U32 pSize)` | Return the largest block can be allocated in a region |

| | |
|---|---|
| STATUS **MemGetTaskUsed**(TASK_ID taskId, P_U32 pSizeUsed) | Memory allocated to the application and the whole system can be inquired about at run time. PPSM-GT returns to the caller the total number of bytes of memory allocated to the task with the given task identifier. |
| U32 **MemGetTotalUsed**(void) | Return the total number of bytes of memory allocated to the whole system. |

## Changing Memory Region

| | |
|---|---|
| STATUS **MemAddRegion**(P_MEM_REGION_ID pRegId, P_U32 startAddr, U32 endAddr) | Add a new memory region based on specified inputs. |
| STATUS **MemDelRegion**(MEM_REGION_ID regionId) | Delete the specified memory region. |
| STATUS **MemResetRegion**(MEM_REGION_ID regionId) | MemResetRegion() resets the memory region to initial settings and can only be used to reset regions allocated by MemAddRegion(). |
| STATUS **MemResizeRegion(**MEM_REGION_ID regionId, U32 endAddr) | MemResizeRegion() allows the memory region to be resized to the new size. The resizing will fail if the region specified is larger than the malloc pool. The new size can be smaller or larger the the current size. The content in the region is kept, allowing developers to resize the region after the system is booted up. Developers are responsible for ensuring that the trimmed-down part is useless. |

As mentioned in the section “Defining Memory Regions,” memory regions are defined and added dynamically in applications. Thus, with APIs such as MemMallocFrom() and MemCallocFrom() where a region is required, the region must be first defined and added. MemAddRegion() adds the region based on the addresses provided, and MemDelRegion() deletes the specified region.

The system returns only errors for the following cases:

- endAddr is zero
- endAddr is even
- endAddr is within +/-24 bytes of current ending address
- endAddr overlaps with another existing address
- For an endAddr that is smaller than the current one, endAddr is not inside a free memory block

## Summary

Memory management services in PPSM-GT reduce the work of memory management to simple API calls. Memory can be arranged into regions and dynamically defined and added during runtime. Memory can also be allocated, released or reallocated dynamically during runtime from the main malloc pool. APIs are available to check the memory usage and status of the malloc pool.

PPSM-GT memory management services provide an effective and easy way of accessing the system, but they do not and cannot guarantee the effective use of the system. To ensure proper operation and the effective use of system memory, system integrators should observe practices of proper memory usage, such as releasing unused memory back to the malloc pool and not requesting too much or too little memory than is necessary.

## Code Examples

Listing 9.1 shows how to allocate, free, reallocate and inquire about memory with PPSM-GT APIs. The example is not intended to solve any particular programming problem and should be used as a reference. It should not be used directly in the application without modification.

**Listing 9.1 Examples of allocating, freeing, reallocating and inquiring about memory**

```
        /*    Variable declaration*/

        TASK_ID MemTask
        P_U8 PstorageA, PstorageB,PstorageC, PstorageD;
        U32 RegionStartAddr, RegionEndAddr;
        MEM_REGION_ID regionId;
        U32 MemoryUsed, MemoryAvail, StackAvail;
        P_U32 pSize
        PLString tempStrN;
        STATUS status;

/* Creating a task call MemTask*/

status = KnlCreateTask(&MemTask,
                          NULL,
                          (const TEXT *)"MEM_TASK",
                          1000,
                          5,
                          USER_MODE);


/* Allocates 256 bytes memory in a default region */

        PstorageA = (P_U8) MemMalloc(256);

/* Allocates memory in a specified region to the target task */

        PstorageB = (P_U8) MemMallocFrom(regionId, 512, MemTask);

/* Allocates memory in a default region to the calling task and
initialize memory to zero. */

        PstorageC = (P_U8) MemCalloc(256);

/* Allocates memory for the specified region to the target task
and initialize memory to zero. */

        PstorageD = (P_U8) MemCallocFrom(regionId, 512, MemTask);
```

```
/* Free the memory reserved of PstorageA and PstorageB*/

        MemFree(PstorageA);
        MemFree(PstorageB);


/* Reallocate the memory of Pstorage C to PstorageA from 256 to
512 bytes*/

        PstorageA = (P_U8) MemRealloc(PstorageC, 512);

/* Copying memory from PstorageD to PstorageB*/

        status = MemCopy(PstorageD, PstorageB, 512)

/* Inquiring Memory status*/

        status= MemGetAvailSize(regionId, MemoryAvail);

        StackAvail = MemGetAvailStack();

        status = MemGetOrgRegionSize(regionId, pSize);

        status = MemGetTaskUsed(MemTask, &MemoryUsed);

        MemoryUsed = U32 MemGetTotalUsed();

      }
```

Listing 9.2 shows how to change the memory region. It shows how to add, delete, reset and resize a memory region. The example is not intended to solve any particular programming problem and should be used as a reference. It should not be used directly in an application without modification.

**Listing 9.2 Examples of changing memory region**

```
        /*    Variable declaration*/
        P_MEM_REGION_ID RegionPtr;
        U32 RegionStartAddr, RegionEndAddr;
        MEM_REGION_ID regionId;
        PLString tempStrN;
        STATUS status;

        status = MemAddRegion(RegionPtr, RegionStartAddr,
RegionEndAddr);

        if( status == SYS_OK)
        {
          sprintf(tempStrN, "Add Reg 0x%x Success \n",
*RegionPtr);
        }
        else
          sprintf(tempStrN, "Add Reg failed\n");

        else if(DeleteRegion)
        {
        status = MemDelRegion(regionId);

        if( status == SYS_OK)
        {
        if(regionId == MEM_ALL_EXT_REGION)
            sprintf(tempStrN, "Del all ext. regions Success\n");
          else
            sprintf(tempStrN, "Del Reg 0x%x Success\n",
regionId);
        }
        else
        {
        if(regionId == MEM_ALL_EXT_REGION)
            sprintf(tempStrN, "Del all ext. regions failed\n");
          else
            sprintf(tempStrN, "Del Reg 0x%x failed\n", regionId);
        }

      else if(ResetRegion)
        {
        status = MemResetRegion(regionId);

        if( status == SYS_OK)
        {
        if(regionId == MEM_ALL_EXT_REGION)
            sprintf(tempStrN, "Reset all ext. regions
Success\n");
```

```
          else
          sprintf(tempStrN, "Reset Reg 0x%x Success\n",
regionId);
        }
        else
        {
        if(regionId == MEM_ALL_EXT_REGION)
            sprintf(tempStrN, "Reset all ext. regions failed\n");
          else
            sprintf(tempStrN, "Reset Reg 0x%x failed\n",
regionId);
        }

      }
        else if(ResizeRegion)
        {
        status = MemResizeRegion(regionId, RegionEndAddr);

        if( status == SYS_OK)
              sprintf(tempStrN, "Resize Reg 0x%x Success\n",
regionId);
        else
              sprintf(tempStrN, "Resize Reg 0x%x failed\n",
regionId);

      }
```

# 10 Power Management Services

PPSM-GT utilizes the power control module of DragonBall™ microprocessors to implement a set of power management tools to achieve system power savings.

Power management services enable applications to:

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

Applications can choose to:

- control the system's power management features directly, or
- use PPSM-GT's automatic power management features.

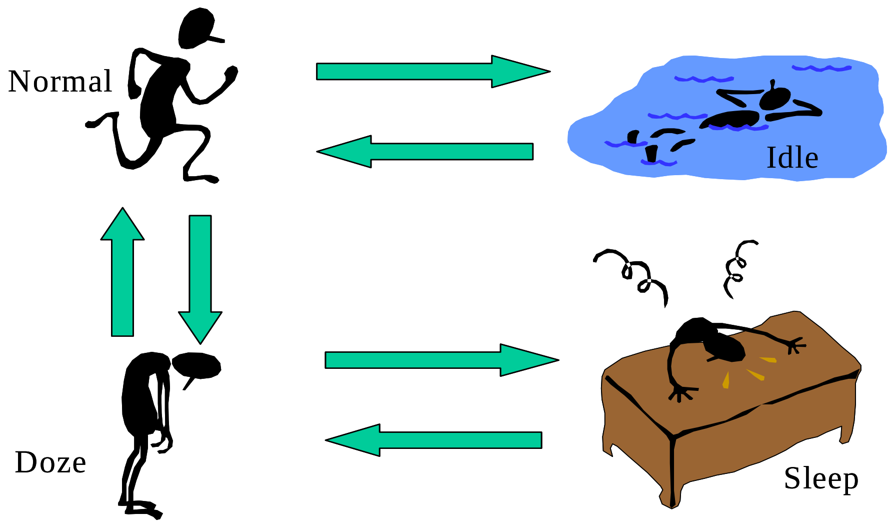
This chapter is organized into the following sections:

- Power Management Fundamentals
- Programming using Power Management Services
- Summary
- Code Example

## Power Management Fundamentals

PPSM-GT supports four types power modes: NORMAL, IDLE, DOZE and SLEEP. For each power mode, PPSM-GT provides a set of APIs for controlling the activities in that power mode. In some of the modes, PPSM-GT allows developers to customize the entry and exit conditions. In general, PPSM-GT controls the switching from one mode to another. Figure 10.1 shows the relationships between the modes. The connections and arrows show the transition from one mode to another. For example, there is no connection between IDLE mode and SLEEP or DOZE. Therefore, the system cannot switch from IDLE mode to DOZE or SLEEP mode directly when in IDLE mode.

**Figure 10.1 Flow of NORMAL, IDLE, DOZE and SLEEP**



**Normal mode**

This is the normal operating mode. In this mode, the CPU core and all the peripherals are active. The system will perform most of its operation under this mode. This is also the most power consuming mode, and the system normally runs in this mode only when it is required.

**Idle mode**

This is one of the power saving modes, wherein the CPU core is turned off while most of the peripherals are active. It is a power saving mode for the system when the static display and the peripherals, but not the core, need to be active.

The system goes to IDLE automatically when there is no activity. The user cannot directly cause the system to go into IDLE mode. PPSM-GT will switch the system from NORMAL mode to IDLE mode when the lowest priority task, the idle task, has executed.

In this mode, no tasks are active; all tasks are either suspended or terminated. Peripherals are, however, still active to received internal or external stimuli. When a stimulus is received, the system will switch from IDLE mode to NORMAL mode.

Users are able to selectively turn off some of the external devices through I/O controls. PPSM-GT provides APIs such as PwrEnterIdle( ) and PwrExitIdle( ) to let developers customize their control of the system when the system enters and exits IDLE mode.

### Doze mode

This is another power saving mode. Only selective modules and devices are off, such as the LCD and LCD controller, and the CPU core runs.

PPSM-GT allows developers to put the system in DOZE mode explicitly or automatically after a certain "idle" period expires. The "idle" period does not mean no activity exists in the system; it means there is no activity for some particular events. In DOZE mode, tasks could be still active, sending "non-wakeup" events to one another; "wakeup" events are absent in DOZE mode. For more information of events, please refer to the chapter on events.

The "idle" period is runtime-configured by the developer. The system software timer keeps track of this period. In both NORMAL and IDLE, the timer is always running. The timer is reset at the end of each wakeup event transmission for a task. This timer stops when the system is in DOZE or SLEEP.

Developers are able to selectively turn off some of the external devices through I/O controls. PPSM-GT provides APIs such as PwrEnterDoze( ) and PwrExitDoze( ) to let developers customize their control of the system when the system enters and exits DOZE mode.

### Sleep mode

This mode saves the most power: the CPU core and all the peripherals except the real time clock module are turned off.

PPSM-GT will switch the system to SLEEP mode automatically when there is no activity in DOZE mode. It is not directly controllable by the user. It is based on the lowest priority task, the idle task, which runs whenever no other task is running. It implies that there is no activity needed to be performed by the system when the idle task is running. Hence, the idle task stops the PLL and CPU core when it runs.

Developers are able to selectively turn off some of the external devices through I/O controls. PPSM-GT provides APIs such as PwrEnterSleep( ) and PwrExitSleep( ) to let developers customize their control of the system when the system enters and exits SLEEP mode.

In SLEEP mode, only an external interrupt will wake up the system to go into DOZE and then NORMAL mode.

### Relationships between the 4 power modes

Figure 10.2 shows the relationships between the power modes. In the beginning the system is in normal mode with the LCD display active.

The system will remain in this mode as long as there are wakeup events such as pen inputs or task activities. In the absence of both, the system will execute the idle task, which will switch the system to IDLE mode.

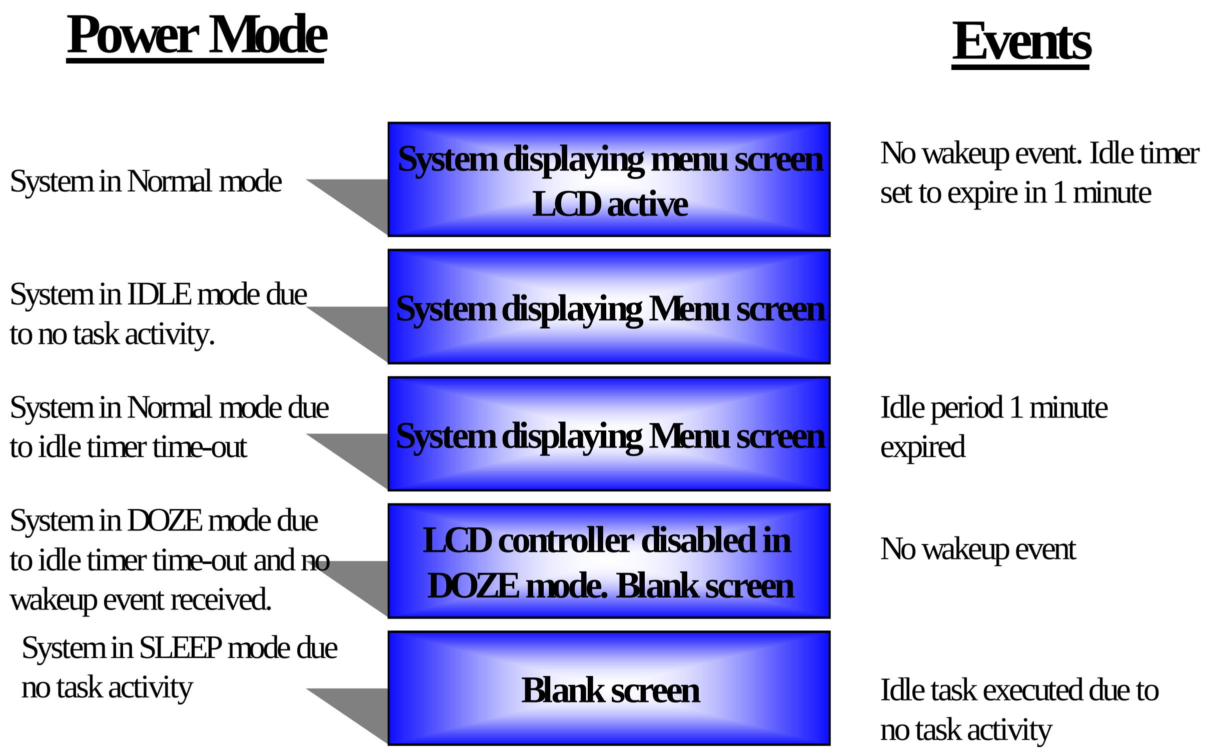
In IDLE the idle timer is still active. Upon the expiration of the idle timer, the system will have NORMAL mode to handle the idle timer time-out. As the idle timer will time out only in the absence of wake up events, the system then goes in DOZE mode on the expiration of the idle timer.

In DOZE mode, the system will switch back to NORMAL mode if there are wakeup events. The system will remain in DOZE mode if there are non-wakeup events that represent the presence of task activities. In the absence of that, the system goes into SLEEP mode, and stays in that mode until an external interrupt occurs to switch it back to DOZE and to NORMAL mode.

## The Idle Task

The idle task is the lowest priority task and runs only when no other tasks are able to run. In addition to turning off the CPU core, the task takes other actions to ensure minimum power consumption.

Figure 10.2 Showing relationship among the power modes


Power Mode
Events
System in Normal mode
System displaying menu screen LCD active
No wakeup event. Idle timer set to expire in 1 minute
System in IDLE mode due to no task activity.
System displaying Menu screen
System in Normal mode due to idle timer time-out
System displaying Menu screen
Idle period 1 minute expired
System in DOZE mode due to idle timer time-out and no wakeup event received.
LCD controller disabled in DOZE mode. Blank screen
No wakeup event
System in SLEEP mode due no task activity
Blank screen
Idle task executed due to no task activity

# Programming using Power Management Services

## Inquiring Power Information

| | |
|---|---|
| U32 **PwrGetDeviceStatus**(VOID) | It returns status of the device |
| POWERMODE **PwrGet-Mode**(VOID) | It returns the power mode: NOR-MAL, IDLE, DOZE or SLEEP |
| U32 **PwrGetSysClk**(VOID) | It returns value of the system clock |

## Controlling DOZE mode

Doze mode control APIs are provided to control the entry into and exit from DOZE. Entry into DOZE mode can be automatic or direct by command. DOZE mode can also be disabled if necessary.

| | |
|---|---|
| STATUS **PwrSetMode**(POWERMODE mode) | Set system to NORMAL or DOZE mode directly. |
| U16 **PwrGetIdleTime**(VOID) | It returns value of the idle time before going into DOZE mode. |
| U8 **PwrIsIdleEnable**(VOID) | It returns value of the idle timer is enable. |
| STATUS **PwrRestartIdle**(VOID) | It resets and restarts the idle timer. |
| STATUS **PwrSetIdleTime**(U16 second) | It sets the idle timer to the time parameter in seconds. |
| STATUS **PwrNotifyDoze**(TASK_ID taskId) | This causes the system to send an event to the task when the system goes into DOZE mode. The event message type is EVT_POWER_GODOZE. |
| STATUS **PwrDisnotifyDoze**(TASK_ID taskId) | This causes the system not to send an event to the task when the system goes into DOZE mode. |

## Disabling DOZE mode when a task is running

### Controlling I/O devices in IDLE power mode

| | |
|---|---|
| STATUS **PwrStopIdle**(VOID) | It will stop the idle timer. |
| U32 **PwrGetIdle**(VOID) | Returns those devices that are set when the system is going into or exiting IDLE mode. |
| VOID **PwrSetIdle**(U32 devices) | Executes the routines provided when the system is going into or exiting IDLE mode. |

The PwrSetIdle() API, when called, executes the routines provided when the system is going into or exiting IDLE mode. PPSM-GT provides only a function call structure, and the routines to execute during the function call must be provided by the system integrator or software developer. In other words, if you want to turn off I/O Port A during IDLE mode, call PwrSetIdle() in the application routines and then fill in the codes to turn off port A in the file pwrdev.c under the PortEnterIdle() section.

PPSM-GT provides a 32-bit variable for easy access of the I/O devices. The LSB, bit 1, has been used by the system for PWR_MC68328_PLL to control SDRAM self-refresh mode. The rest of the 31 bits, from bit 2 to bit 32, are open for definition. At the end of this chapter, Listing 10.1 illustrates the definition of bit 2 and its use to turn off the LCD.

## Controlling I/O devices in DOZE and SLEEP power modes

| | |
|---|---|
| VOID **PwrGetExitDoze**(VOID) | It returns those devices that are set when system is exiting DOZE mode |
| VOID **PwrSetEnterDoze**(U32 devices) | This routine will execute when system is going into DOZE mode and set those devices that are specified in the pwrdev.c |

| | |
|---|---|
| VOID **PwrSetExitDoze**(U32 devices) | This routine will execute when system is exiting DOZE mode and set those devices that are specified in the pwrdev.c |
| U32 **PwrGetEnterSleep**(VOID) | It returns those devices that are set when system goes into SLEEP mode |
| U32 **PwrGetExitSleep**(VOID) | It returns those devices that are set when system is exiting SLEEP mode |
| VOID **PwrSetEnterSleep**(U32 devices) | This routine will execute when system is going into SLEEP mode and set those devices that are specified in the pwrdev.c |
| VOID **PwrSetExitSleep**(U32 devices) | This routine will execute when system is exiting SLEEP mode and set those devices that are specified in the pwrdev.c |

The controlling I/O devices in DOZE and SLEEP modes work on the same fundamental principles as in the IDLE mode. Please read the section “Controlling I/O devices in IDLE power mode” and the examples in detail to understand the operating principles. The only difference is that instead of having a single API to control entry into and exit from the power mode, in DOZE and SLEEP modes two APIs are provided: one for entering and the other for exiting.

As a rule of thumb, the net sum of devices that are turning off in DOZE and SLEEP mode and devices that are turning on in DOZE and SLEEP mode must be equal to avoid any system malfunction. For example, if 3 devices are turned off in DOZE mode and 4 devices are turned off in SLEEP mode, the total number of devices turned off is 7. Therefore, if 2 devices are turned on when switching the system into DOZE from SLEEP mode, then when the system switches into NORMAL mode, 5 more devices must be turned on. This is, however, not a “MUST HAVE” rule imposed by the system. If the system integrator or the software developer fully understands the design and knows what he is doing, this rule need not be obeyed.

## Summary

Power management services provide a means to control the power modes supported by the DragonBall microprocessors. There are two ways to set the control of power mode switching: automatically or directly. For automatic switching, the system will switch to the power mode after a time-out period expires. In direct control, the system will switch the power mode after the command has been received.

APIs are also provided for I/O controls during power mode switching. There are device driver routines, and the system integrators have to provide the content for controlling the devices during power mode switching.

## Code Example

**Listing 10.1 Controlling LCD in DOZE mode**

```
file PwrLCD.h
/******Header File Includes ******************/

/* Devices control bit */
#define PWR_LCD         0x0002


file Pwrcheck.c
/****** Routine to turn off port A in Idle ***/

#include PwrLCD.h

.
.
/**** LCD to be turned on/off when out/in DOZE mode ***/
PwrSetEnterDoze(PWR_LCD);
PwrSetExitDoze(PWR_LCD);
.
.

file pwrdev.c
/************************************************/
void PortEnterDoze(U32 devices)
{
    /*  Check that bit PWR_LCD is selected then turn off LCD  */
    if (devices & PWR_LCD)
    {
        TurnOffLCD( );
    }

    return;
}

/***********************************************/
void PortExitDoze(U32 devices)
{
    /*  Check that bit PWR_LCD is selected then turn on LCD  */
    if (devices & PWR_LCD)
    {
        TurnOnLCD( );
    }

    return;
}
```

### Example 1 explanation

There are, altogether, 3 files in Listing 10.1: `PwrLCD.h`, `Pwrcheck.c`, and `Pwrdev.c`.

The file PwrLCD.h is the header file that stores the bits' definition of the devices. There are a total of 32 bits, and bits 2 to 32 are open for usage.

```
#define PWR_LCD        0x0002
```

This line means that bit 2 is used for LCD power control. To avoid a double definition, do not duplicate the bit definition.

The file Pwrcheck.c is a user application program that calls PwrSetEnterDoze()/PwrSetExitDoze() to set the devices to be controlled when the system enters and exits DOZE mode. In the example, the device to be controlled is LCD.

```
PwrSetEnterDoze(PWR_LCD);
PwrSetExitDoze(PWR_LCD);
```

The file Pwrdev.c is a device driver file, and the internal routines such as PortEnterIdle() are device driver routines that are called just before the system enters DOZE mode. The system integrator or the software developer must provide codes to gain control of the devices. In the simple example above, the LCD is turned off.

```
if (devices & PWR_LCD)
{
TurnOffLCD();
}
```

# 11 System Application Services

System application services are PPSM-GT core services and are for laying the foundation of the whole program. Like the foundation of a building, they are important and can never be ignored. System application services are mainly used in the main program to set up the program environment, such as input and display media for the whole program. The concept of an *application* is uniquely defined in PPSM-GT and is covered in this chapter.

This chapter is organized into the following sections:

## Application Fundamentals

An application sets the stage for tasks to perform actions. This analogy of an application to the stage in a drama helps to illustrate what an application is in PPSM-GT. It creates the task operating environment.

In a drama, the stage is where everything take place; without the stage there is no drama. With the stage alone, there is also no drama. A drama needs actors and actresses with different roles. Using this analogy for PPSM-GT, the application is where things happen, and an application needs tasks to give it meaning.

An application is a conceptual element and it represents the look and feel of a particular function. Taking a calculator application as an example. The digits are displayed on the screen when the icons/keys representing the digits are touched, and the result is displayed on the LCD screen when addition or subtraction is performed. The application is like a black box which has icons/keys inputs and LCD screen as output. The handling of inputs and generating of output on the LCD screen are all done by software routines know as tasks. The application is the place that enable the activities to happen.

There are 5 major elements in the application architecture: Application, Panning Screen, Input Context, Graphic Context and Task. The following section provides an introduction to these elements and the relationship among them.

## Panning Screens

Panning screens are memory buffers that store the display content, and they are independent entities that need to be created when required. Panning screens are created by AppCreatePanScreen( ) or AppCreateGC( ). AppCreateGC( ) will create a panning screen that is bound to the GC, whereas AppCreatePanScreen( ) creates an independent panning screen.

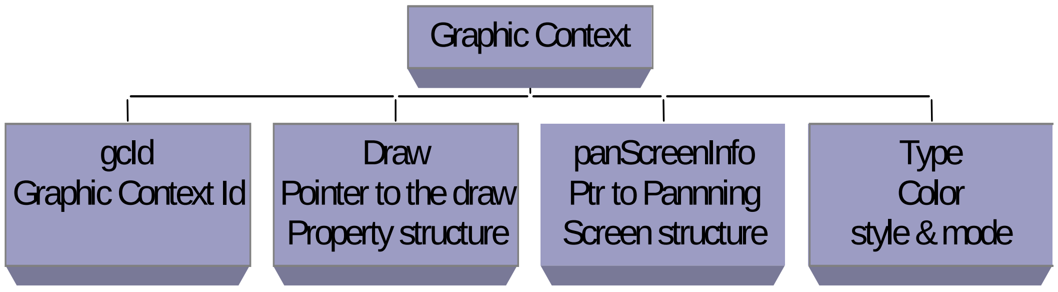
Normally, one application needs one panning screen if the application needs to display any content on the LCD display. Therefore, in a multiple application environment, each application may need to have its own panning screen for display.

In the multiple panning screen environment, at any one time only one panning screen can be mapped to the LCD display. This panning screen is known as the “active panning screen” and it’s contents will be display on the LCD display. For the rest of the panning screens, its’ contents will not appeared on the LCD display until the panning screen is mapped onto the LCD display.

The number of panning screens supported is defined by the user and restricted by the system memory.

## Graphic Context

**Figure 11.1 Graphic Context Structure**



### What is Graphic Context

Graphic Contexts, GCs are memory buffer that stored the drawing property for a task.

All the PPSM-GT graphic routines will refer to the task current graphic context to get information like dot width, fill pattern, color and style. Having the above information, they will then draw the image on the particular panning screen that binds to the graphic context.

All the graphic routine would refer to the property of the draw structure in the graphic context. If the draw pointer in the graphic context is NULL, the graphic routine will use the default value for its drawing.

The default value would be used upon draw structure creation, user would change those properties by the graphic API provided.

By default the drawing properties are set to

| | | |
|---|---|---|
| • DEFAULT_DOTWIDTH | = | 1 |
| • DEFAULT_PATTERN | = | 0 |
| • DEFAULT_BACKCOLOR | = | WHITE |
| • DEFAULT_BORDER | = | 1 |
| • DEFAULT_SPACE | = | 0 |

As shown in Figure 11.1 - Graphic Context Structure, it consists of the Graphic Context Id, display color, style, mode, the pointers to the draw property,and pointers to panning screen and the current task Id that use the GC.

Graphic contexts are independent entities and have to be created separately if required using GpxCreateGC( ).

## Input Context

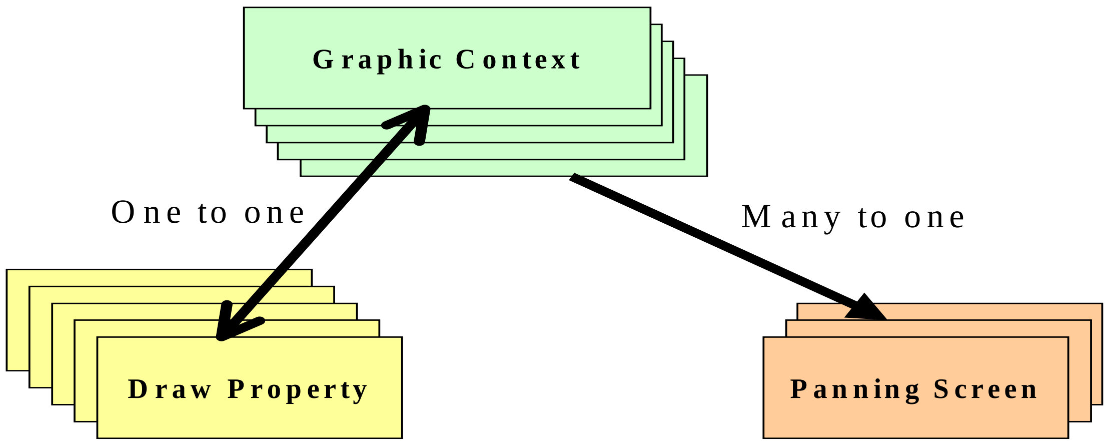
Input context are memory buffer that store pen input time-out, sampling rate, pen size, pen color, active area list, and task that is bound to the input context. Applications are tied to their corresponding input context that defining the touch screen behavior, unique to an application. An input context defines what is to be carried out upon a particular pen action. For example, a pen touch on an icon may trigger a particular response from the application.

## Task

Task are software routines that does the actual work. While the input context defines the responses to pen input activities, all "actual work" are done by underlying tasks. For example, when a particular icon is touched, an event will be sent to the task that serves that icon. When lines are drawn in an input area (within which contour of pen touch, in addition to pen up and down actions, is recorded), the coordinates of the lines are sent to a particular task for processing (for example, character recognition).

## Relationship of GC and Panning Screen and Drawing property

**Figure 11.2 Relationship of Draw Property, Panning Screen and Graphic Context**



The draw property and panning screen relationship with graphic context is as shown in Figure 11.2 shows the relationship of the GC & Panning Screen. Draw property and graphic context has a one to one relationship upon creation. When a GC is created, the draw property list is also created. The draw property belongs to the GC and will be deleted when the GC is deleted. The content of the draw property could be modified with the following APIs:

- GpxSetColor() to change the background color
- GpxSetDotWidth() to change the dot width
- GpxSetPatternFill() to change the pattern

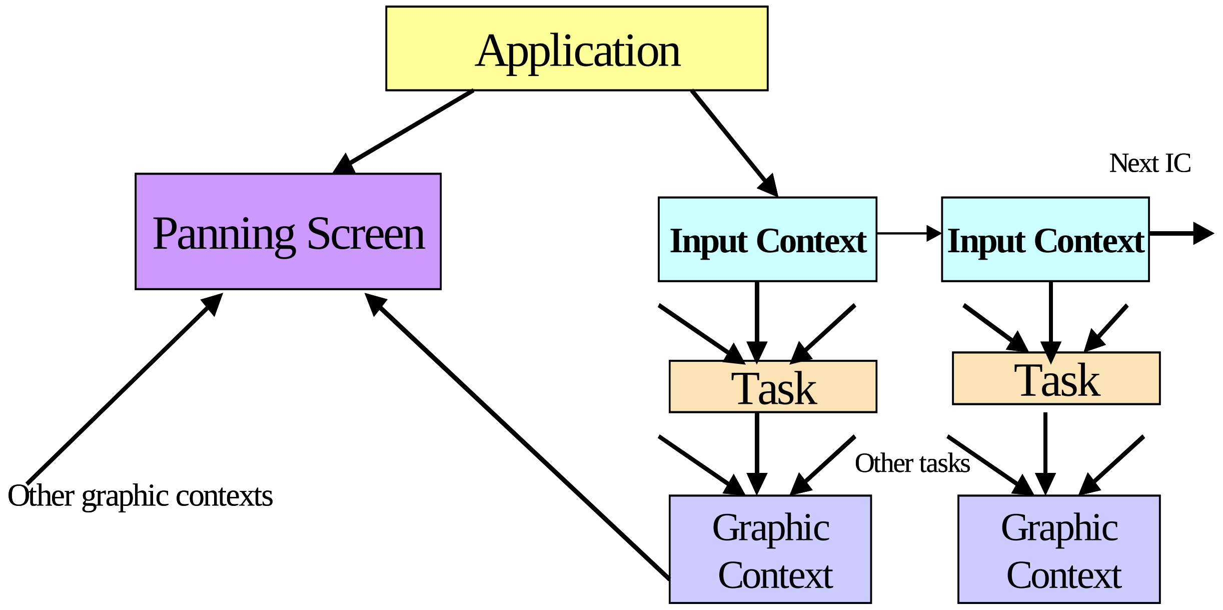
On the other hand, the relationship between panning screen and graphic context is different. It is a one to many relationship, i.e at any one time one GC can be bind to one or no panning screen, but one panning screen is not limited to one GC. Many GCs could be pointing to one panning screen just like many different pens are used for drawing on the same piece of paper. The relationship is a also a dynamic one such as at anytime the panning screen could be change with AppSetCurrPanScreen( ) or AppSetPanScreen( )

Panning screen could be created with AppCreatePanScreen() or together with GC creation. By the default, no panning screen is created when graphic context is created.

Therefore a graphic context will have a draw property and may or may not have a panning screen. The effect have not having panning screen will be discussed in the following section.

## Relationship of task, application, panning screen and graphic context

**Figure 11.3 Relationship of Task, Application, Panning Screen & Graphic Context**



### Task and Graphic context

Graphic context,GC works with task for drawing purpose, and one task can have only one GC at any one time, but one GC is not limited to one task. That is many tasks are allowed to share GC. This allows many tasks to draw on the same panning screen with same drawing property just like many people are using the same pen to draw on the same piece of paper. PPSM-GT keeps track of the number of tasks using the GC.

This number will increased by 1 if additional task are using the GC, and decreased by 1 when task unbind itself with the GC. Bind and unbind are done by KnlBindGC( ). To bind the GC, the GCId is required. To unbind the GC, the value ’0’ is in place of the GcId.

The system will not allows, any task to delete the GC if it is still in use. Therefore if GC is no longer required by any task, it should be unbind.

### Task, Panning Screen and Graphic context

Tasks used draw property on GC to draw graphics onto the panning screen that is bind to the GC. To draw on different panning screen, the task could either switch the panning screen on the GC or switch the GC totally. The difference is switching GC, changes the draw property if the two GCs draw properties are different.

### Application, input context, task, panning screen and graphic context

Applications are tied to their corresponding input context which defines the touch screen behavior and is unique to an application. Panning screen defines where the output goes. An active panning screen will have it’s contents displayed and seen on the LCD display. Figure 11.3 shows the relationships of application and the rest of the elements in the application architecture, and defines an application’s input/output behavior by it’s relationship with the list of input contexts and a panning screen. While a particular input context is solely owned by an application, a panning screen can be shared by multiple applications.

Basically, every application performs it’s function in a loop of

- Taking input with the input context(s)
- Perform necessary actions with tasks serving the input context(s). This may include write back of result to the panning screen by the tasks with parameters defined in their corresponding graphic context.

A particular task can serve multiple number of input contexts, thus may serve more than one application. Graphic context, in similar way, can serve multiple number of tasks, thus may also serve more than one application

In such an environment, which application is going to respond to the inputs from the multiple input context? Which panning screen will have it’s context displayed on the LCD display. Such questions are addressed and answered by the application known as “the current one”.

In the multitasking environment, all applications when created are active until they are terminated. Some applications may have tasks actively going some activities, other might be actively waiting. An application therefore becomes “the current one” when it is intentionally switched on by the designers using AppSwitch(). Therefore, the decision on which application should be “the current one” at any one time lays in the hand of the designers and not the system.

When an application is switched on by AppSwitch() command, it becomes “the current one”. All the pen input activities is send to it’s corresponding input context to determine what action is to be carried out, and it’s corresponding panning screen is mapped to the LCD display.

Only one application will be "the current one" at a particular time, although tasks serving other applications may be running in the background.

## Multiple applications environment

A multiple applications environment is normally used in a complex system where a single system environment is unable to handle the system requirements. In such a system, a developer could use multiple applications to handle different system requirements as long as the following rules are followed:

1. Each application must have its own panning screen, graphic context and input context.
2. AppSwitch() is used to switch from one application to another.

## Entry and Exit callback functions

Entry and Exit callback functions are routines that are executed before and after application switching, respectively.

In application creation, users can fill in the Entry and Exit callback functions. When an application is to be swapped in, its entry function will be executed first. When an application is to be swapped out, its exit function will be executed before the swap out. For example:

VOID SchedularEntryCallback() and VOID SchedularExitCallback()

# Programming using System Application Services

There are four groups of system application services and are listed as followed:

- Programming Task Operating Environment
- Programming Graphic Context
- Programming Input Context
- Programming Input Context

## Programming Task Operating Environment

### Creating Application

| | |
|---|---|
| STATUS **AppCreate**(P_APP_ID pAppId, P_TEXT pName, P_VOID entryCallback, P_VOID exitCallback, U16 iconWidth, U16 iconHeight, P_U8 pIconImage, U16 ringBufferSize) | It creates the application with specified parameters like the name of the application, entry callback function, exit callback function, and so on. |

### Deleting Application

| | |
|---|---|
| STATUS **AppDelete**(APP_ID appId) | It deletes the application specified by AppId. This routine cannot be used to delete the download application. |

### Getting Application ID

| | |
|---|---|
| STATUS **AppGetAppIdFromIC**(IC_ID icId, P_APP_ID pAppId) | Retrieves the application ID from Input context ID while AppGetCurrent() returns the current application ID. If there is no current application, it returns NULL. |
| APP_ID **AppGetCurrent**( ) | Get the current Application ID. |

### Getting Icon Info

| | |
|---|---|
| STATUS **AppGetIcon**(APP_ID appId, P_U16 pIconWidth, P_U16 pIconHeight, P_U8 * pIconImage) | It returns the application icon with its width and height in pixels. |

### Getting Application name

| | |
|---|---|
| STATUS **AppGetName**(APP_ID appId, P_TEXT * pName) | It returns the pointer of the application name. |

### Getting Next Application on the App List

| | |
|---|---|
| STATUS **AppGetNext**(APP_ID appId, P_APP_ID pAppId) | It returns the next application pointer. This applies to both download applications and general applications. |

### Getting Previous Application

| | |
|---|---|
| STATUS **AppGetPrev**(APP_ID appId, P_APP_ID pAppId) | It searches the applications in the general application list and returns the previous application pointer for the specific application. |

### Switching Application

| | |
|---|---|
| STATUS **AppSwitch**(APP_ID appId) | It switches the application to the specific application and changes the LCD display correspondingly. If the new application has no panning screen, no change will be seen in LCD. |

## Programming Graphic Context

### Creating Graphic Context

| | |
|---|---|
| STATUS **AppCreateGC**(P_GC_ID pGCId, U16 horz, U16 vert) | AppCreateGC() creates the graphic context that is requested, and the horz and vert parameters are for creating a binding panning screen. If no panning screen is required, set the two parameters to zero. |

### Deleting Graphic Context

| | |
|---|---|
| STATUS **AppDeleteGC**(GC_ID gcId) | AppDeleteGC() deletes the GC as specified by the gcId. |

## Programming Input Context

### Adding Input Context

| | |
|---|---|
| STATUS **AppAddIC**(APP_ID appId, IC_ID icId) | This routine inserts an input context to the front of an input context list of the application . |

### Getting first Input Context in list of App

| | |
|---|---|
| STATUS **AppGetFirstIC**(APP_ID appId, P_IC_ID pIC) | It gets the first input context in the input context list in the application. |

### Moving Input Context to the top of App’s IC list

| | |
|---|---|
| STATUS **AppMoveICToTop**(APP_ID appId, IC_ID icId) | This routine moves the specific input context to top of the application’s input context list |

### Removing Input Context from App’s IC list

| | |
|---|---|
| STATUS **AppRemoveIC**(IC_ID icId) | This routine removes the input context from its application’s input context list |

### Swapping the App’s IC list

| | |
|---|---|
| STATUS **AppSwapICList**(APP_ID appId, IC_ID newIC, P_IC_ID pIC) | This routine swaps the input context list in specific application and return the old one to calling routine |

## Programming Panning Screen

### Creating Panning Screen

| | |
|---|---|
| STATUS **AppCreatePanScreen**(P_SCREEN_ID pScreenId, U16 horz, U16 vert) | AppCreatePanScreen() and AppDeletePanScreen() are for creating and deleting panning screens, respectively. When creating the panning screen, the dimension of the panning screen (horz and vert) must be provided. The following horz and vert inputs will cause an error, and the system will not create the panning screen.<br>1. Zero value.<br>2. Multiplier of 4 for pixels 4 design, multiple of 8 for pixels 2 design and multiple of 16 for pixel 1 design.<br>3. Value cannot exceed the maximum permissible values supported by the hardware. Refer to section on LCD controller in hardware manual. |

### Binding Panning Screen

| | |
|---|---|
| STATUS **AppBindPanInfo**(APP_ID appId, SCREEN_ID panInfo) | Bind the panning screen to the application specified by the application ID. |

### Changing Panning Screen

| | |
|---|---|
| STATUS **AppSetPanScreen**(GC_ID gcId, SCREEN_ID screenId) | Set the panning screen to the graphic context provided |
| STATUS **AppSetCurrPanScreen**(SCREEN_ ID screenId) | Set the panning scrren to the current graphic context |

Two possible effect could be achieved with changing panning screen:

– Direct all graphics output to off-screen

– Direct all graphic output to new on-screen

#### *Direct all graphic output to off-screen*

Run-time computation intensive image generation and display could be slow. Users may see the graphics output appears slowly on the LCD display screen. By using GpxSetPanScreen() allows applications to direct all output from PPSM-GT 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 LCD screen using AppBindPaninfo(). This will give the effect that the image is displayed instantaneously.

GpxSetPanScreen() assumes that all input parameters are valid. If screenId is zero, no panning screen will be attached to that graphic context.

GpxSetCurrPanScreen() works like GpxSetPanScreen() except for the absent of GcId as the input parameter. Therefore GpxSetCurrPanScreen, will set the current panning screen specified by the current screenId to the active GcId.

#### *Direct all graphic output to new on-screen*

When required to change the current active panning screen to another new active panning, GpxSetPanScreen() and AppBindPaninfo() are used together.

The effect would cause the application to switch from the old panning screen to the new panning screen immediately. Whatever that is drawn on the new panning screen after the AppBindPaninfo() will appear on the LCD screen.

### Deleting Panning Screen

| | |
|---|---|
| STATUS **AppDeletePanScreen**(SCREEN_ID screenId) | Delete the panning screen specified by the screenId. |

### Deleting Panning Screen Info from App list

| | |
|---|---|
| STATUS **AppRemovePanInfo**(APP_ID appId) | It sets the panning screen information in the specific application to NULL. |

### Getting Panning Screen Info

| | |
|---|---|
| STATUS **AppGetPanInfo**(APP_ID appId, SCREEN_ID * pPanInfo) | It returns the pointer of the panning screen information for the specific application. |

### Get Panning Screen Width

| | |
|---|---|
| U16 **AppGetPanScreenWidth**(void) | GpxGetPanScreenWidth( ) returns to the caller the panning screen width, in terms of pixels, of the current application. |

### Get Panning Screen Height

| | |
|---|---|
| U16 **AppGetPanScreenHeight**(void) | GpxGetPanScreenHeight( ) returns to the caller the panning screen height, in terms of pixels, of the current application. |

### Get Current Panning Screen Id.

| | |
|---|---|
| STATUS **AppGetCurrPanScreen**(P_SCREEN_ID pScreenId) | The routine returns the panning screen ID from the graphic context of the current task or from the current application. |

**Get Panning Screen Id.**

| | |
|---|---|
| STATUS **AppGetPanScreen**(GC_ID gcId, P_SCREEN_ID pScreenId) | The routine returns the panning screen ID from the specified graphic context. |

## Summary

Applications have a unique definition in the PPSM-GT system. Metaphorically speaking, they are like a stage on which tasks perform actions. In the PPSM-GT environment, developers can use one application or multiple applications to design the system. It all depends on the complexity of the system and the creativity of the design.

Application defines the behavior resulted from the interaction among the underlying elements (input context, panning screen, task and graphic context). The relationship of these elements with an application is shown in Figure 11.3 - Relationship of Task, Application, Panning Screen & Graphic Context

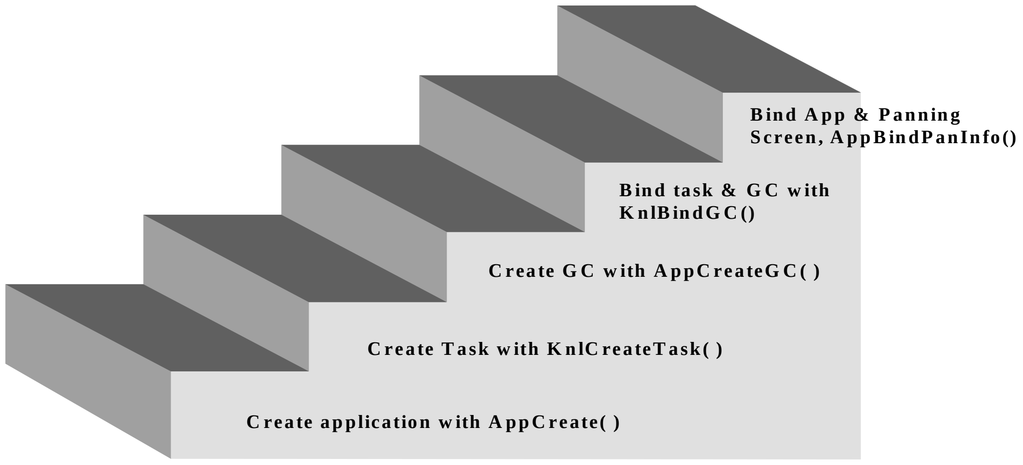
PPSM-GT application services can handle both the simple one-application system or the complex multi-application system. In addition to multi-application design, developers can have multiple panning screens with a single application or multiple applications. The combinations are many, and the key is creativity. Therefore, to unleash the power of PPSM-GT, focus effort on the design of the system and devise a combination that works for the system.

## Code Example

### Setting up the application environment

Typically in PPSM-GT programming, the main module is the module that sets up the application environment for the rest of the system. The following example describes the step-by-step approach of a main program that sets up the application environment.

**Figure 11.4 Typical Steps for Creating an Application**


Bind App & Panning Screen, AppBindPanInfo()
Bind task & GC with KnlBindGC()
Create GC with AppCreateGC( )
Create Task with KnlCreateTask( )
Create application with AppCreate( )

## Setting up an application

1. First create an application for the whole system. This will set up the environment for the rest of the tasks to operate in. The system will assign a AppId to the application if AppCreate is successful.
2. Create all the main tasks in the current application, such as UI application tasks, network tasks, and SCI tasks. All tasks created are suspended till the system is ready for operation.
3. Create the graphic context with a panning screen for all graphic applications. The graphic context needs to be bound to the task, and the panning screen needs to be bound to the application.

**Listing 11.1 Example of setting up an application without application framework**

```
/* Variable Declaration*/
#define DEFAULT_MODE 1
#define NORMAL_PRIORITY 6
#defnine HIGH_PRIORITY 10
#define HIGHEST_PRIORITY 15

P_APP_IDpAppId;
TASK_IDTaskAId, TaskBId, TaskCId;
GC_ID gcId;
SCREEN_ID PanScreenId;
const TEXTTaskNameA= {’T’,’A’,’S’,’K’,’_’,’A’,0};
const TEXTTaskNameB= {’T’,’A’,’S’,’K’,’_’,’B’,0};
```

```
  const TEXTTaskNameC= {’T’,’A’,’S’,’K’,’_’,’C’,0};
  TEXT    AppName[] = {’A’,’P’,’P’,0};
  STATUS  status;

STATUS main(void)

  /* Create an application name APP with 3000 bytes ring buffer*/
    status = AppCreate(&gpAppId, AppName, 0, 0, 0, 0, 0, 3000);

    /* Create task A with normal priority*/
status = KnlCreateTask( &TaskAId, (P_VOID) TaskNameA,
        NORMAL_PRIORITY,DEFAULT_MODE, 4000);
      KnlSuspend(TaskAId);

  /* Create task B with high priority*/
    status = KnlCreateTask( &TaskBId, (P_VOID)TaskNameB,
HIGH_PRIORITY, DEFAULT_MODE,4000);
      KnlSuspend(TaskBId);

  /* create task C with highest priority*/
  status = KnlCreateTask( &TaskCId, (P_VOID)TaskNameC,
HIGHEST_PRIORITY, DEFAULT_MODE,4000);
      KnlSuspend(TaskCId);

    /* create graphic context for application environment with a
panning screen of 160 x 240*/
    AppCreateGC(&gcId, 160, 240);

  /* bind the graphic context to the task*/
    KnlBindGC(gpTaskId, gcId);

    /* get the panning screen from graphic context*/
    GpxGetPanScreen(gcId, &PanScreenId);

  /* bind panning screen to the application*/
    AppBindPanInfo(AppId, PanScreenId);

  /* Start executing application and activitate task A.*/
    AppSwitch(AppId);
    KnlResume(TaskAId);
}
```

**Listing 11.2 Use GpxSetPanScreen() to draw image**

```
SCREEN_ID newScreen;
APP_ID testApp
GC_ID gpxcont;

/* Direct all graphic routine to memory area
pointed by
   new panning screen to newScreen */

AppSetPanScreen(gpxcont, newScreen);
GpxSetDotWidth(6);
GpxSetColor(BLACK);
GpxSetStyle(REPLACE_STYLE);
GpxDrawRec(20,80,100,140,0);
AppBindPaninfo (testApp, newScreen);

:
```

# 12 System Event Management Services

System events consist of information that is sent in PPSM-GT's core from one task to another. The information can be sent from a task or a device driver. There are two groups of system events in PPSM-GT: normal and broadcast events. Each group can be further divided into 4 types, as follows:

- Erasable and wake up events
- Erasable and non-wake up events
- Non-erasable and wake up events
- Non-erasable and non-wake up events

## Channels

Channels are the media on which the information is carried. Only tasks on the same channel can receive the event being broadcast.

## Event ports

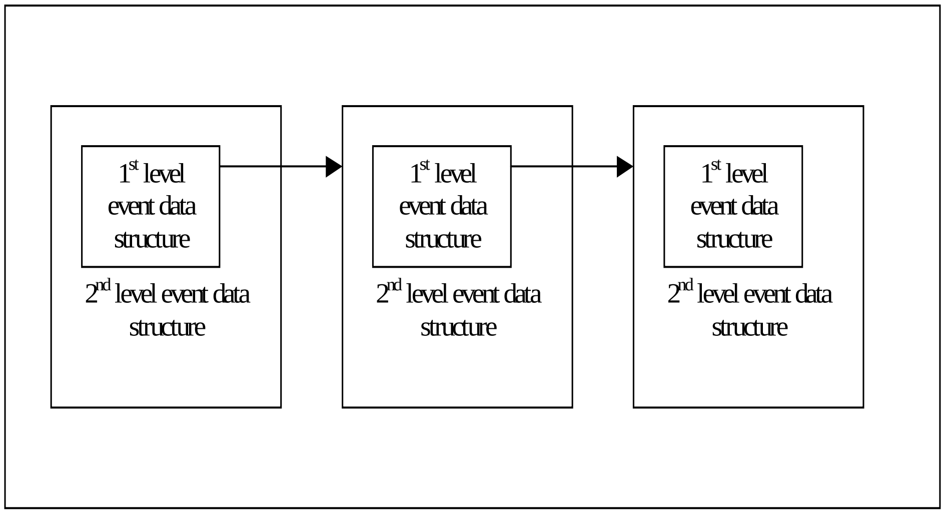
Event ports are the connection points for the task. In order to receive an event properly, a task must create an event port and bind the created port to itself using EvtCreatePort() and KnlSetCurrEventPort(). One task can have only one event port.

Event ports are created and bound to a task by default when a task is created.

Event ports are not required for a task sending an event.

## Event data structure

**Figure 12.1 Level event data structure**


1st level event data structure
2nd level event data structure
1st level event data structure
2nd level event data structure
1st level event data structure
2nd level event data structure

All the events have a one- or two-level based structure. The first level is a general data structure called “EVENT,” and it includes a pointer to another “EVENT” identity, event type and source.

**Listing 12.1 A first-level event**

```
typedef struct _EVENT
{
    struct _EVENT   *next;      /* Pointer to next event */
    EVTTYPE         type;       /* Event type */
    TASK_ID         source;     /* Source of event */
   U16            usageCount; /* Number of task using this event
*/
} EVENT, *P_EVENT;
```

The second level is an extension on the first level. It may be defined upon any particular type of event, and users may extend it by defining their own second-level data structure. This is for the users to send customized messages from one task to the other.

**Listing 12.2 Setting up a customized second-level event**

```
typedef struct _SHORT_EVT
{
EVENT   event;              /* 1st level event */
U16     data;               /* 2nd level event, 16-bit data */
} SHORT_EVT, *P_SHORT_EVT;
```

PPSM-GT automatically cleans up the event data structure after a user has received it. For memory allocated and pointed to by a level 2 event data structure, the user needs to perform the cleanup manually.

## Event Headers

Event headers are event identifiers that are attached to the event when the system sends out an event to the task. They describe the type of event and are sent as event types in the event data structure. The event header file "EVT_typ.h" shows the list of system event types that PPSM-GT supports. System integrators or software developers should check the event against the event header to determine the action required. The file "EVT_typ.h" may be changed in later versions of PPSM-GT.

System integrators or software developers could also send a user-defined event from one task to another. To send a user-defined event, call EvtAllocType() to allocate and get an available event type from the system.

Not all services require the system to send event messages to a task. For example, graphic manipulation and text management services do not need to send any event message to a task, so there is no event type defined.

### Decoding a received event

All events with a number larger than or equal to EVT_USR_DYNAMIC_BASE and smaller than 0x3FFF are considered to be user-defined event types.

**Table 12.1 System Event Header Description Table**

| Type Definition | Description |
|---|---|
| EVT_NONE | No activity |
| EVT_USR_DYNAMIC_BASE | User-defined event start |
| **Kernel Event** | No system event defined |
| **Event Management Event** | No system event defined |
| **Memory Management Event** | No system event defined |
| **Text Management Event** | No system event defined |
| **Graphic Manipulation Event** | No system event defined |
| **Software Timer Event** | No system event defined |
| **Alarm Event** | |
| EVT_ALM_TYPE | Type of alarm |
| EVT_ALM_EXPIRED | Alarm expired |
| **Sci Management Event** | |
| EVT_UART1 | UART data |
| EVT_UART2 | UART data |
| **Audio Management Event** | |
| EVT_AUD_OFF | Audio off |
| EVT_AUD_WAVEINUSE | Audio wave in use |
| EVT_AUD_TONEINUSE | Audio tone in use |
| EVT_AUD_MELODYINUSE | Audio melody in use |
| EVT_AUD_MELODYPAUSE | Audio melody pause |
| **Power Management Event** | |
| EVT_POWER_IDLE | Idle timer |
| EVT_POWER_GODOZE | Go sleep |
| EVT_POWER_OUTDOZE | Exit sleep |
| **Application Event** | |
| EVT_APP_DOWN_ID | Application download ID |
| **Pen Input Event** | |
| EVT_PEN_ICON_TOUCH | Icon pen touch |
| EVT_PEN_ICON_DRAG | Icon drag in |

| Type Definition | Description |
|---|---|
| EVT_PEN_ICON_UP | Icon pen up |
| EVT_PEN_ICON_DRAG_OUT | Icon drag out |
| EVT_PEN_INPUT_TOUCH | Input area pen touch |
| EVT_PEN_INPUT_DRAG | Input area pen drag in |
| EVT_PEN_INPUT_UP | Input area pen up |
| EVT_PEN_INPUT_DRAG_OUT | Input area drag out |
| EVT_PEN_INPUT_DATA | Input area pen coordinate |
| EVT_PEN_KEY_TOUCH | Key area pen touch |
| EVT_PEN_KEY_DRAG | Key area pen drag in |
| EVT_PEN_KEY_UP | Key area pen up |
| EVT_PEN_KEY_DRAG_OUT | Key area drag out |
| EVT_PEN_KEY_DATA | Key area data |
| EVT_PEN_KEY_TIMEOUT | Key area pen up timeout |
| EVT_PEN_TIMEOUT | Pen up timeout |
| **SKY Event** | |
| EVT_SKY_KEY | Soft keyboard |
| EVT_SKY_BEGIN_REPEAT | Soft keyboard Begin Repeat |
| EVT_SKY_REPEAT | Soft keyboard repeat |
| **INP Event** | |
| EVT_INP_CHAR | HWR a char |
| EVT_INP_PL_CHAR | HWR a char for PL |
| EVT_INP_ADDPAD | Add input pad |
| EVT_INP_DELPAD | Delete input pad |
| EVT_INP_UNINSTALL | Uninstall engine |
| EVT_INP_TIMEOUT | Timeout |
| **IrDA Event** | |
| EVT_IRD_FRM_RECV | IrDA framer received a frame available for processing |
| EVT_IRD_FRM_SENT | IrDA framer has send out a packet |

| Type Definition | Description |
|---|---|
| EVT_IRD_TIMEOUT | IrDA framer timeout |
| EVT_IRD_MEDIACHANGE | |
| EVT_IRD_IRCOMME_WRITE | Ircomm: data can be written |
| EVT_IRD_IRCOMME_READ | Ircomm: data is available to be read |
| EVT_IRD_IRCOMME_CLOSE | Ircomm: the close is complete |
| EVT_IRD_IRCOMME_STATUS_CHANGE | Ircomm: the control status changed |
| EVT_IRD_IRCOMME_STATUS_SENT | Ircomm: status was sent to peer |
| EVT_IRD_IRCOMME_WRITE_ERR | Ircomm: write failed |
| EVT_IRD_OBCE_CONNECTED | Obex: TinyTP connection has been established for the client |
| EVT_IRD_OBCE_DISCONNECT | Obex: Underlying IR connection has been disconnected |
| EVT_IRD_OBCE_DISCOVERY_FAILED | Obex: Device discovery failed to find an OBEX capable device |
| EVT_IRD_OBCE_COMPLETE | Obex: requested OBEX operation is complete |
| EVT_IRD_OBCE_ABORTED | Obex: current OBEX client operation has been aborted |
| EVT_IRD_OBCE_HEADER_RX | Obex: server received a header |
| EVT_IRD_OBSE_RX_IND | Obex: server indicated receiving a header and object body |
| EVT_IRD_OBSE_RX_COMPLETE | Obex: server completed receiving header and object body |

## Types of events

There are 4 types of events in PPSM-GT, and they are as follows:

- Erasable and wake up events
- Erasable and non-wake up events
- Non-erasable and wake up events
- Non-erasable and non-wake up events

### Erasable and Non-erasable Events

Erasable events are events that the system will automatically delete when no more tasks are using them. An event is considered not to be in use when its “usageCount” is down to zero. The memory of this event will be deleted.

Non-erasable events are events that the system will not automatically delete when no more tasks are using them. The memory of this event will not be deleted; manual deletion of the memory is required.

The memory pointed to by a pointer in the second-level user-defined data structure cannot be deleted automatically. It has to be deleted by a user manually.

### Wake up and non-wake up Events

A wake up event, when sent, will restart the idle timer (see Power Management) in NORMAL mode or wake up the system in SLEEP / DOZE SLEEP mode. To set up a wake up event, developers may call EvtSetWakeup() after initializing and before sending an event.

A non-wake up event, when sent, will not restart the idle timer (see Power Management) in NORMAL mode, nor will it wake up the system in SLEEP/DOZE SLEEP mode. To set up a non-wake up event, developers may call EvtSetUnwakeup() after initializing and before sending an event.

An event defaults to being a non-wake up event after calling EvtInitEvent().

### Broadcast event

Broadcast events are events that are sent to more than one task. In order for tasks to receive the broadcast events, the tasks must be connected to the broadcast channel. There are two classes of broadcast events in PPSM-GT: timely and untimely.

- Timely broadcast events: Send out the event to all the tasks in the same channel but skip those that are suspended or waiting semaphore. There is a user-defined time limit. The event is removed from the channel after a user-defined duration even when some of the tasks in the channel did not receive the event.
- Untimely broadcast events: Send out the event to all the tasks in the same channel but skip those that are suspended or waiting semaphore. There is no time limit.

### Broadcasting Channel

A channel contains a list of tasks ordered according to their priority. There could be many broadcast channels in the system, and tasks have to subscribe to the individual broadcast channel to receive the broadcast events. A task can be in more than one channel at the same time.

A broadcasting event contains two parts. They are the broadcast data structure and the event that needs to be sent. They form a broadcasting event to be sent out.

The broadcasting event may be reused after broadcasting by checking the “usageCount” of the received event to determine if any task is using the event or not. The checking can be done with EvtGetUsage().

### Time broadcasting

A time field is used to monitor the live time of the broadcasting event. Once the live time has expired, a task that has received this broadcasting event will take the event to its event queue and remove the broadcasting event from the channel.

# Programming Using Event Management Services

## Creating an Event

| | |
|---|---|
| STATUS **EvtInitEvent**( P_EVENT pEvent, EVTTYPE type) | An event needs to be created before it can be sent. To create an event, first allocate memory with MemMalloc() for the event size and initialize the event data structure with the EvtInitEvent() API. |
| EVTTYPE **EvtAllocType**(VOID) | It dynamically allocates a free event type; if not, it returns EVT_NONE. |
| STATUS **EvtFreeType**(EVTTYPE) | It frees a dynamically allocated event type. |
| STATUS **EvtRegisterType**(EVTTYPE) | It registers and requests a particular event type from the system. |

EvtInitEvent() will initialize the event data structure fields to:

- “next” is a pointer to next event, to NULL.
- “type” is the type of event, to user-defined value “type.”
- “source” is used to indicate which task sent out the event, to 0.
- “usageCount” is used to count the usage of an event, to 0.

The value of usageCount is automatically increased when a task is using the event and decreased when the task is no longer using it. This value shall not be altered by a user after the sending out. A user may read this value to check if any task is still using the event by calling EvtGetUsage().

## Setting up Erasable Event

| | |
|---|---|
| STATUS **EvtSetErasable**(P_EVENT pEvent) | To set up an erasable event, use the EvtSetErasable() API after initializing but before sending an event. After initializing an event with EvtInitEvent(), the event defaults to being erasable. |

## Setting up Non-erasable Event

| | |
|---|---|
| STATUS **EvtSetUnerasable**(P_EVENT pEvent) | To set up Non-erasable events, use the EvtSetUnerasable() API before sending an event. The event will not be deleted automatically after being received. |

## Setting Wake up Event

| | |
|---|---|
| STATUS **EvtSetWakeup**(P_EVENT pEvent) | EvtSetWakeup() sets an event be to automatically wake up the system after the sending out. After initializing an event by EvtInitEvent(), the event is defaulted to be non-wake up. |

## Setting up Non-wake up Event

| | |
|---|---|
| STATUS **EvtSetUnwakeup**(P_EVENT pEvent) | EvtSetUnwakeup() sets an event not to automatically wake up the system. |

## Checking the Event Type

| | |
|---|---|
| STATUS **EvtIsErasable**(P_EVENT pEvent) | EvtIsErasable() verifies if an event is auto-erasable. |
| STATUS **EvtIsWakeup**(P_EVENT pEvent) | EvtIsWakeup() verifies if an event is auto-waking the system. |
| BOOL **EvtIsTypeAvailable**(EVTTYPE) | It tests whether an event is available or not. |

## Sending Events

| | |
|---|---|
| STATUS **EvtSend**(P_EVENT pEvent, TASK_ID taskId) | It is the function to send an event to a task. |
| STATUS **EvtSendUrgent**(P_EVENT pEvent, TASK_ID taskId) | It sends an urgent event to a task |

## Getting an event

| | |
|---|---|
| EVTTYPE **EvtCheck**(VOID) | Check for an event in the event queue. The system does not wait for an event if there is no event; it will continue executing the next instruction. |
| EVTTYPE **EvtGet**(VOID) | Check for an event in the event queue. The system will wait for an event if there is no event. The calling task is put to TASK_STATUS_WAITING. |

| | |
|---|---|
| STATUS **EvtWait**(EVTTYPE evtType, TICK milliseconds) | It returns when any one of the following two occurs:<br>1. The specified event is placed in the head of the event queue.<br>2. The time-out interval elapses.<br>All other events are queued in the event FIFO queue. |
| STATUS **EvtWaitMultiple**(<br>U16 numOfType,<br>EVTTYPE* pEvtType,<br>TICK milliseconds,<br>BOOL waitAll) | It returns when any one of the following two occurs:<br>1. Either one or all of the specified events are placed in the head of event FIFO queue.<br>2. The time-out interval elapses.<br>All other events are queued in the event queue. |

A task gets an event from its event queue, which is based on first in first out (FIFO). PPSM-GT provides two APIs to supply asynchronous or synchronous control of the system.

In asynchronous mode, a task is kept running even if it cannot get any event. A task calls EvtCheck() to check and tries to get an event from its event queue. EvtCheck() returns event data type, if any; otherwise it returns EVT_NONE.

In synchronous mode, the rest of the process cannot proceed if a task cannot get any event. A task calls EvtGet() and waits until it gets an event from its event queue. EvtGet() returns event data type, if any.

Because EvtGet() waits for the event, it is not suitable to be used in any interrupt routines.

In both of these modes, the first event in the queue is moved out and treated as the current event in the task. The “usageCount” of the previous event is decreased by one.

## Receiving Event information

| | |
|---|---|
| P_EVENT **EvtGetEvent**(VOID) | EvtGetEvent() returns a pointer to the current event. Returns NULL if no event. |
| EVTTYPE **EvtGetType**(VOID) | EvtGetType() returns the current event type. Returns EVT_NONE if no event. |
| U16 **EvtGetUsage**(P_EVENT pEvent) | EvtGetUsage() returns the number of tasks that are currently using this event. |
| TASK_ID **EvtGetChTask**(CHANNEL_ID channelId, P_U32 pTemp) | EvtGetChTask() returns the first task in a channel if the “pTemp” is NULL. If the input value is a valid address gotten from this API before, it returns the next following task of the channel. |
| TASK_ID **EvtGetChLastTask**(CHANNEL_ID channelId) | EvtGetChLastTask() returns the last task in the channel. |
| TASK_ID **EvtGetChNumTask**(CHANNEL_ID channelId) | EvtGetChNumTask() returns the number of tasks in the channel. |

## Deleting Event information

| | |
|---|---|
| VOID **EvtRmCurrEvent**(VOID) | EvtRmCurrEvent() removes the current event. A user is normally not required to call this, unless he wants to remove the current event from the current task immediately. By default, the current event will be removed automatically when either EvtGet() or EvtCheck() is called. |

| | |
|---|---|
| STATUS **EvtDelInQueue**(EVTTYPE) | To delete all the events with given event type in the queue of the calling task |
| STATUS **EvtFlushQueue**(VOID) | To flush all the events in the queue of the calling task |

## Setting up and deleting the broadcasting structure

| | |
|---|---|
| BRDCST_ID **EvtCreateBroadcast**(VOID) | EvtCreateBroadcast() creates a broadcasting data structure. Returns the broadcastId. |
| U8 **EvtIsBrdcstId**(BRDCST_ID id) | EvtIsBrdcstId() verifies if a number is a Broadcast ID. |
| STATUS **EvtDeleteBroadcast**(BRDCST_ID id) | EvtDeleteBroadcast() deletes a broadcasting data structure. |
| STATUS **EvtRmBrdcstFromCh**(BRDCST_ID brdcstId) | EvtRmBrdcstFromCh() removes a broadcast event from a channel. No operation occurs if it is not on any channel. |

## Setting up and deleting the channel structure

| | |
|---|---|
| CHANNEL_ID **EvtCreateChannel**(VOID) | EvtCreateChannel() creates a channel data structure. Returns the channel ID. |
| U8 **EvtIsChannelId**(CHANNEL_ID id) | EvtIsChannelId() verifies if a number is a channel ID. |
| STATUS **EvtDeleteChannel**(CHANNEL_ID id) | EvtDeleteChannel() deletes a channel data structure. |

## Getting the Broadcast Channel

| | |
|---|---|
| CHANNEL_ID **EvtGetBrdcstChannel**(BRDCST_ID id) | It gets the channel ID of a broadcasting event that is currently on. |

## Adding a Task to the Broadcast Channel

| | |
|---|---|
| STATUS **EvtAddToChannel**(TASK_ID taskId, CHANNEL_ID channelId) | EvtAddToChannel() adds the task to the channel. Tasks that are already on the channel cannot be duplicated. |

## Deleting a Task from the Channel

| | |
|---|---|
| STATUS **EvtRmTaskFromCh**(TASK_ID taskId, CHANNEL_ID channelId) | It removes a task from a channel. |

## Setting up and deleting Event port

| | |
|---|---|
| EVTPORT_ID **EvtCreatePort**(VOID) | EvtCreatePort() creates an event port and returns the event port ID. Port needs to be bound to a task before use. |
| STATUS **EvtDeletePort**(EVTPORT_ID eventPort) | EvtDeletePort() deletes the event port. Port needs to be removed from the task before calling this. |
| Neither routine is suitable for interrupt routines. | |

### Sending Broadcast Events

| | |
|---|---|
| STATUS **EvtSetBrdcstEvent**(BRDCST_ID id, P_EVENT pEvent) | EvtSetBrdcstEvent() configures the broadcasting event. |
| STATUS **EvtSendBrdcstEvent**(BRDCST_ID brdcstId, CHANNEL_ID chId, TICK time) | EvtSendBrdcstEvent() broadcasts an event on a channel. If “time” is EVT_FOREVER, the event will be sent until it reaches the end of the channel. |

### Receiving Broadcast Events

| | |
|---|---|
| P_EVENT **EvtGetBrdcstEvent**(BRDCST_ID id) | It gets the event being sent currently under the broadcasting data structure. |

## Summary

System Event Services are intertask communications. Like telecommunication services in the real world, the system event services provide the means for tasks to communicate with one another. Before any intertask communication can take place, the communication port must first be set up.

Broadcast events are a special kind of event that allow one-to-many task communication. In order for tasks to receive broadcasting messages, they must first subscribe to the broadcast channels.

## Code Examples

The code examples provided are for reference only. They should not be used without modification.

**Listing 12.3 Setting up a Non-erasable event**

```
typedef struct EVENT
{
    struct EVENT   *next;       /* Pointer to next event */
    EVTTYPE         type;       /* Event type */
    TASK_ID         source;     /* Source of event */
   U16            usageCount; /* Number of task using this event
*/
} EVENT, *P_EVENT;

/* Defining two type of event TYPE1 and TYPE2 */
U16 TYPE1 = EvtAllocType();
U16 TYPE2 = EvtAllocType();
P_EVENT     pEvent1;

/* Reserving memory for the event gpEvent1; */
    pEvent1 = (P_EVENT)MemMalloc(sizeof(EVENT));

/* Set up the event structure and declare it as type 1 */
    EvtInitEvent(pEvent1, (EVTTYPE)TYPE1);

/* Set it as non erasable event */
    EvtSetUnerasable(pEvent1);
```

**Listing 12.4 Setting up a non-wake up event**

```
typedef struct EVENT
{
    struct EVENT   *next;       /* Pointer to next event */
    EVTTYPE         type;       /* Event type */
    TASK_ID         source;     /* Source of event */
   U16            usageCount; /* Number of task using this event
*/
} EVENT, *P_EVENT;

/* Defining two type of event TYPE1 and TYPE2 */
U16 TYPE1 = EvtAllocType();
U16 TYPE2 = EvtAllocType();
P_EVENT     pEvent2;

/* Reserving memory for the event gpEvent2; */
```

```
    pEvent2 = (P_EVENT)MemMalloc(sizeof(EVENT));

/* Set up the event structure and declare it as type 2 */
    EvtInitEvent(pEvent2, (EVTTYPE)TYPE2);

/* Set it as non walk-up event */
    EvtSetUnwakeup(pEvent2);
```

**Listing 12.5 Setting up a broadcast event**

```
typedef struct EVENT
{
    struct EVENT   *next;       /* Pointer to next event */
    EVTTYPE         type;       /* Event type */
    TASK_ID         source;     /* Source of event */
    U16            usageCount; /* Number of task using this event
*/
} EVENT, *P_EVENT;
    BRDCST_ID BroadcastId;
    CHANNEL_ID ChannelId;
    TASK_IDTaskAId;
    const TEXTTaskNameA= {'T','A','S','K','_','A',0};
    STATUS status;


/* Creating a task to receive the software timer event*/
    status = KnlCreateTask(&TaskAId,
        (P_VOID)TaskA, TaskNameA, 4000, 6, 1);

/* Create the broadcast channel*/
    BroadcastId = EvtCreateBroadcast();
    ChannelId = EvtCreateChannel();

/* Add task A to channel for transmitting & receiving broadcast */
    status = EvtAddToChannel(TaskAId, ChannelId);

/* Set and Send broadcast event for 10 millseconds*/
    status = EvtSetBrdcstEvent(BroadcastId, pEvent2);
    status = EvtSendBrdcstEvent(BroadcastId, ChannelId, 10);
```

13

# Software Timer Handling Services

Software Timer Handling services are a set of software routines that operate the software timer that is based on the software reference timer. PPSM-GT uses a 32-bit register as a software reference timer. Every count is 1 tick, and the range of the software reference timer is from 0 to $2^{32}$ ticks.

A tick is a system-developer-defined time unit. It is a division of time and is normally specified in milliseconds. In PPSM-GT, a tick is defined as 1 millisecond.

The reference timer defaults as free running and counts all the time.

The software timer functions can act as an alarm, and the alarm takes its reference from the software reference timer. The resolution is in milliseconds, and the range is 0 to 24 hours. PPSM-GT provides APIs to control, set and use this alarm.

This chapter is organized into the following sections:

## Software Timer Handling Fundamentals

Software Timer Handling Services APIs provide access to the software timer. They consist of routines to create a new software timer and routines to reconfigure an existing timer. Once a software timer is created, it should be reused when possible by reconfiguration to conserve system resources.

### Basic Steps for Setting Up Software Timer

Table 13.1 shows the basic steps required for setting up a new software timer using PPSM-GT.

**Table 13.1 Basic steps for new Software Timer**

| Steps | Description |
|---|---|
| Step 1 | Before using the software timer, the user needs to determine the number of counts required. The resolution of the software timer is approximately 1 millisecond; therefore, a 1-second timer is 1000 counts, 1 minute is 60,000 counts, and so on. One millisecond per tick is an approximate value. To get a more accurate time, use SwtGetResolution, which returns the number of counts per second, and then calculate the exact count required. |
| Step 2 | Create a new software timer using SwtCreate( ). This software timer has not been configured. PPSM-GT will return a software timer ID. Refer to this ID to configure and manipulate the timer. |
| Step 3 | Configure the new software timer using SwtInitTimer( ). The input parameters required by this API are SWT ID, task ID, count, reload, pointer to event, size of event, function pointer and pointer to argument. |
| Step 4 | After the timer has been set up, it is ready to be used. SwtStartTimer( ) will start the software timer running. The timer will count until the count specified by “count” expires, and an event will be sent to the task specified in the software timer data structure. If a function is specified, then the function will be executed with the argument provided. |
| Step 5 | The active timer can be stopped by SwtStopTimer( ) if necessary. |
| Step 6 | A timer that are not used should be deleted with SwtDelete( ). A timer need not be stopped to be deleted. |

## Programming Using Software Timer Handling Services

### Creating the Software Timer

| | |
|---|---|
| SWT_ID **SwtCreate**(VOID) | It creates an software timer that is not configured and returns the software timer ID. |

### Initializing the Software Timer

| | |
|---|---|
| STATUS **SwtInitTimer**(SWT_ID swtId, TASK_ID taskId, TICK count, TICK reload, P_EVENT pEvent, U16 size, P_VOID func, U32 arg) | It initializes a software timer by setting up the software timer data structure. Set parameters that are not required to “0.” |

### Starting the Software Timer

| | |
|---|---|
| STATUS **SwtStartTimer**(SWT_ID timer) | Start the configured software timer. |

### Stopping the Software Timer

| | |
|---|---|
| STATUS **SwtStopTimer**(SWT_ID swtId) | Stop the active software timer and reset the timer to 0. |

### Deleting the Software Timer

| | |
|---|---|
| STATUS **SwtDelete**(SWT_ID swtId) | Delete the software timer. |

## Configure the existing software timer

| | |
|---|---|
| STATUS **SwtSetEvent**(SWT_ID swtId, P_EVENT pEvent, U16 eventSize) | It sets up the event pointer onto the software timer structure. The event needs to be created first. Refer to System Event Management Services for details on creating an event. |

## Setting up the count on the software timer

| | |
|---|---|
| STATUS **SwtSetCount**(SWT_ID swtId, TICK count) | It sets up the count onto the software timer structure. |

## Setting up the function on the software timer

| | |
|---|---|
| STATUS **SwtSetFunc**(SWT_ID swtId, P_VOID func, U32 arg) | It sets up the function onto the software timer structure. When the software timer expires, the function will execute with the given argument.<br>Users need to provide the function that handle the argument. If no argument is required for the function, the function can ignore the argument. |

## Setting up the Argument on the software timer

| | |
|---|---|
| STATUS **SwtSetArg**(SWT_ID swtId, U32 arg) | It sets up the pointer to the argument that was used by the function. This is an additional option for users who need only to set the argument for the same function. |

## Setting up the task on the software timer

| | |
|---|---|
| STATUS **SwtSetTaskId**(SWT_ID swtId, TASK_ID taskId) | It sets up the taskId onto the software timer structure. When the timer expires, the event will be sent to this taskId. |

## Restarting stopped software timer or refreshing active software timer

| | |
|---|---|
| STATUS **SwtRestartTimer**(SWT_ID swtId, TICK count) | It restarts the stopped software timer or refreshes the timer if it is active. |

## Reading Software timer data structure

| | |
|---|---|
| STATUS **SwtGetCount**(SWT_ID swtId, P_TICK pCount) | It gets the count for a software timer. |
| P_EVENT **SwtGetEvent**(SWT_ID swtId, P_U16 pSize) | It gets the event pointer and its size for a software timer. |
| U32 **SwtGetResolution**(VOID) | It gets the number of ticks per second. |
| STATUS **SwtGetTaskId**(SWT_ID swtId, P_TASK_ID pTaskId) | It gets the taskId for a software timer. |
| U8 **SwtIsInUse**(SWT_ID swtId) | It tests if a given SWT ID is in use or not. |
| U8 **SwtIsSwtId**(SWT_ID swtId) | It tests if a number is a SWT ID or not. |

## Reading and checking the reference Software timer

| | |
|---|---|
| TICK **SwtReadRefTime**(VOID) | It returns the reference timer value. |
| TICK **SwtDiffRefTime**(TICK beginTime, TICK endTime) | It returns the difference between two given reference time values. This routine takes care of wrap-around situations. The actual difference between the start and end time shall not be bigger than half a wrap. |

## Summary

The software handling services provide access to the software reference timer through the APIs. Developers do not need to write the software timers in the application; they are already written. To use the software timers, developers just need to create, configure, and use them. After a software timer is used, it can either be deleted or kept for future use.

## Code Example

The following example shows how to use the software timer handling services to create 3 software timers, and, based on the key input, the timers are restarted or stopped. The code provided is merely for reference. It does not solve any specific problems and cannot be used directly without being modified first.

**Listing 13.1 Creating and Using Three Software Timers**

```
  /* Variable Definition*/
  #defineNORMAL_PRIORITY 6
  #define DEFAULT_MODE 1
/* Defining two type of event TYPE1, TYPE2 & TYPE3*/
  #define     TYPE1        EVT_USER_BASE + 1
  #define     TYPE2        EVT_USER_BASE + 2
  #define     TYPE3       EVT_USER_BASE + 3
  TASK_IDSWTTaskId;
  const TEXTSWTTaskName= {’S’,’W’,’T’,’T’,’A’,’S’,’K’,0};
  const TEXTTaskNameB= {’T’,’A’,’S’,’K’,’_’,’B’,0};
  const TEXTTaskNameC= {’T’,’A’,’S’,’K’,’_’,’C’,0};
  P_U32 TaskCode[] = {
    (P_VOID)SWTTask,
    (P_VOID)TaskB,
    (P_VOID)TaskC
    };
  EVENT SWTEvt[3];
  SWT_IDswtId[3];
  EVTTYPE SwtType1

  int i;
  STATUS status;
  typedef struct EVENT
  {
    struct EVENT   *next;       /* Pointer to next event */
    EVTTYPE         type;       /* Event type */
    TASK_ID         source;     /* Source of event */
   U16             usageCount; /* Number of task use this event */
  } EVENT, *P_EVENT;


/* Reserving memory for the event gpEvent1; */
  for (i=0; i<3; i++)
      SWTEvent[i] = (P_EVENT)MemMalloc(sizeof(EVENT));

/* Set up the event structure and declare it as type 1 */
    EvtInitEvent(&SWTEvt[0], TYPE1);
    EvtInitEvent(&SWTEvt[1], TYPE2);
    EvtInitEvent(&SWTEvt[2], TYPE3);

/* Set it as non erasable event */
      for (i=0; i<3; i++)
        EvtSetUnerasable(SWTEvt[i]);

/* Creating a task to receive the software timer event*/
    status = KnlCreateTask(&SWTTaskId,
          (P_VOID)SWTTask, SWTTaskName,
          4000, NORMAL_PRIORITY, DEFAULT_MODE);
  if (status == SYS_OK)
  {
    KnlSuspend(SWTTaskId);
  }

/* Creating the 3 software timer */
  for (i=0; i<3; i++)
    {
        swtId[i] = SwtCreate();
    }
/* Setting up the 3 software timer */
    SwtInitTimer(swtId[0], SWTTaskId, 10, 0, &SWTEvt[0],
sizeof(EVENT), 0, 0);
    SwtInitTimer(swtId[1], SWTTaskId, 50, 0, &SWTEvt[1],
sizeof(EVENT), 0, 0);
    SwtInitTimer(swtId[2], SWTTaskId, 20, 0, &SWTEvt[2],
sizeof(EVENT), 0, 0);
  }

        :
        :
        :
        :

/* Example of restarting Timer S1 and print the message S1 :
Active */
         {
          SwtRestartTimer(swtId[0], 10);

         }

/* Example of stopping SWT timer S3*/

        SwtStopTimer(swtId[2]);
        SwtDelete(swtId[2]);

  }

//return SYS_OK;
}
```

Section 5

# Developing with Application Services

The chapters in this section help answer the question, “How do you use the Application Services in the system?” When used, Application Services provide additional features to your system. They are not essential services; in their absence, the system can still operate.

Each chapter has three parts.

- The first part of each chapter discusses System Services fundamentals. It introduces the concepts that must be understood before using the services.
- The second part of each chapter explains the APIs: the interfaces and functions of each API. For details of each API, please refer to the PPSM-GT API reference document.
- The third part of each chapter consists of a short summary and code example that shows how to use the APIs. Please note that the example mainly shows how to use a particular API and is not designed to address any specific problem. The examples should not be copied and used blindly.

The chapters in this section are as follows:

- Chapter 14, “Real Time Clock Handling Services”—introduces Real Time Clock routines that have been created in the PPSM-GT library. It provides example of how to use the APIs for the Real Time Clock Services.

- Chapter 15, “Alarm Services”—introduces alarm routines that have been created in the PPSM-GT library. It provides examples of how to use the APIs for the Alarm Services.
- Chapter 17, “Audio Management Services”—introduces Audio routines that have been created in the PPSM-GT library. It provides examples of how to use the APIs to generate music with Audio Services.
- Chapter 18, “Serial Communication Interface Services”—introduces the SCI services. It covers how to program and use the SCI to transmit and receive information through the serial port. It also provides information on how to handle multiple SCI communications.
- Chapter 19, “IrDA Management Services”—introduces the IrDA services. It covers how to program and use the IrDA to transmit and receive information through the IrDA link.
- Chapter 20, “Networking Services”—introduces the fundamentals of TCP / IP services. PPSM-GT supports the BSD socket for transport layer programming, and the APIs in this chapter has been deliberately kept similar to the standard socket naming convention to avoid confusion when doing socket programming.

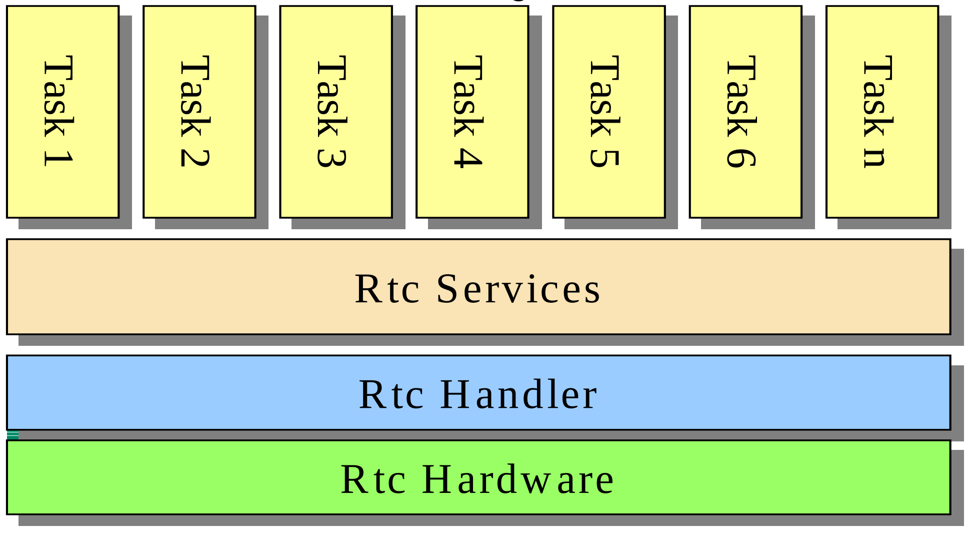
14

# Real Time Clock Handling Services

The PPSM-GT RTC handling services are provided to handle the RTC modules of the DragonBall family of microprocessors. The APIs enable ease of use, including checking real time clock information.

This chapter is organized into the following main sections:

**Figure 14.1 RTC Services Block Diagram**



## RTC Fundamentals

Figure 14.1 shows the RTC structure. There are 3 main components:

- The RTC Hardware block is the RTC module in a DragonBall microprocessor.
- The RTC handler block handles all RTC-related information, including from an API block. It will filter the information before passing the information to the Hardware RTC block. It performs functions including the following:
  - processes information from APIs
  - validates the information for valid second, minute, hour, day, month, and year
  - calculates leap years
  - computes the day of week
  - updates RTC hardware
- The RTC Handling Services API block interfaces with the application for RTC information. The gathered information is handled over to the RTC handler block for processing.

# Programming Using RTC Handling Services

## Checking Leap Year

| | |
|---|---|
| void **RtcIsLeapYear**(U16 year, P_U8 leapyear) | RtcIsLeapYear( ) is for checking the leap year. It checks the input year to be checked for a leap year, and it returns a Boolean expression: TRUE if the input year is a leap year and FALSE otherwise. |

## Getting RTC Information

These APIs are for getting the current time, date, time and date, and day of the week. No inputs are required except for the day of the week, when the date is required. The outputs are current time, date, time and date, or day of the week, depending on the API.

| | |
|---|---|
| STATUS **RtcGetTime**(P_U8 hour, P_U8 minute, P_U8 second) | RtcGetTime( ) returns the values of the second, minute and hour of the current time into the buffer provided. |
| STATUS **RtcGetDate**(P_U16 year, P_U8 month, P_U8 day) | RtcGetDate( ) returns the values of the year, month, and day for the current date into the buffer provided. |
| STATUS **RtcGetDateTime**(P_U16 year, P_U8 month, P_U8 day, P_U8 hour, P_U8 minute, P_U8 second) | RtcGetDateTime( ) returns the values of the year, month, and day for the current date and second, minute and hour of the current time into the buffer provided. |
| STATUS **RtcGetDayofWeek**(U16 year, U8 month, U8 day, P_U8 dayofweek) | RtcGetDayofWeek( ) returns the day of the week based on the input date. |

## Setting the Time and Date

The date and time can be set in PPSM-GT using RtcSetTime( ), RtcSetDate( ), and RtcSetDateTime( ). PPSM-GT will validate the inputs. If the date or time inputs are invalid, the system will not proceed to set the date or time and will return a SYS_ERR error message.

The default time at power reset is at 0:00:00 hour, 1 January 2000.

| | |
|---|---|
| STATUS **RtcSetTime**(U8 hour, U8 minute, U8 second) | RtcSetTime( ) sets the second, minute and hour of the real time. It returns SYS_ERR for invalid time inputs. |
| STATUS **RtcSetDate**(U16 year, U8 month, U8 day) | RtcSetDate( ) sets the year, month, day of the RTC date. It returns SYS_ERR for invalid date inputs. |
| STATUS **RtcSetDateTime**(U16 year, U8 month, U8 day, U8 hour, U8 minute, U8 second) | RtcSetDate( ) sets the year, month, and day for the current date and the second, minute and hour of the RTC. It returns SYS_ERR for invalid time or invalid date inputs. |

## Validating the Time and Date

| | |
|---|---|
| STATUS **RtcValidTime**(U8 hour, U8 minute, U8 second) | For simple verification of a valid time. |
| STATUS **RtcValidDate**(U16 year, U8 month, U8 day) | For simple verification of a valid date. |

RtcValidTime( ) and RtcValidDate( ) are provided for simple verification of a valid time and date, respectively. The inputs are the time or date in question, and the outputs are the system status.

For a valid time or date, the system status is SYS_OK.

For an invalid time or date, the system will return the error message SYS_ERR.

## GMT Time

To faciliate the concept of world time calculation, APIs are added to include the concept of time zone into PPSM-GT. This concept is especially useful in email applications where each outgoing email is stamped with the local time sent plus the GMT offset.

For example, for an email sent on Friday, April 20, 2001, from Hong Kong (GMT+08:00) at a local time of 14:25:07, the following header will be sent along: "Date: Fri, 20 Apr 2001 14:25:07 +0800". This stamping allows the recepient email application to display the sent time in its side's local time.

Using the same example, if the recepient is in Tokyo (GMT+09:00), his email application will interpret the sent time as Friday, April 20, 2001, 15:25:07.

| | |
|---|---|
| STATUS **RtcGetGMTOffset**(P_S8 Gmtoffset) | Returns the pointers for GMT offset, The GMT offset string is of format "sHH00" where s is the sign (+ or -) and HH the number hour offset from GMT (1 to 13 if s is +; 1 to 12 if s is -; 0 regardless of s) |
| STATUS **RtcSetGMTOffset**(S8 Gmtoffset) | Set the GMT offset The GMT offset string is of format "sHH00" where s is the sign (+ or -) and HH the number hour offset from GMT (1 to 13 if s is +; 1 to 12 if s is -; 0 regardless of s).<br><br>By defaulted tthe offset is "+0800" |
| STATUS **RtcGetGMTime**(P_U16 year, P_U8 month, P_U8 day, P_U8 hour, P_U8 minute, P_U8 second) | RtcGetGMTime( ) returns the value of the year, month, and day for the current date and second, minute and hour of the current time in the GMT format including the offset into the buffer provided. |

## Summary

The RTC Handling Services provide APIs for checking, getting, validating and setting real time clock information. The RTC module runs most of the time, even when the device is in sleep mode, and it supports time, date and day features. The RTC module also supports the alarm function in PPSM-GT.

## Code Example

This example shows how the RTC services are used in an RTC application.

**Listing 14.1 RTC Services in an RTC Application**

```
STATUS RTCTaskApp()
{
    U16             year;
    U8              date[6];
    char            TimeString[12];
    char            DateString[18];
    year = mDate.GetYear();
    date[0] = ‘29’;
    date[1] = ‘12’;
    date[3] = ‘12’;
    date[4] = ‘30’;
    date[5] = ‘10’;
   RtcSetDateTime (year, date[0], date[1], date[3], date[4],
date[5]);
  while(1)
  {
    RtcGetDate(&year, &date[0], &date[1]);
    RtcGetTime(&date[3], &date[4], &date[5]);
    sprintf(DateString, "%d / %d / %d",year,date[0],date[1]);
    sprintf(TimeString, "%d : %d : %d",date[3],date[4],date[5]);
        if(DatePane )
        {
           DatePane -> SetString(DateString);
           TimePane -> SetString(TimeString);
        }
        PLSleep(500);
  }
  return SYS_OK;
}
```

15

# Alarm Services

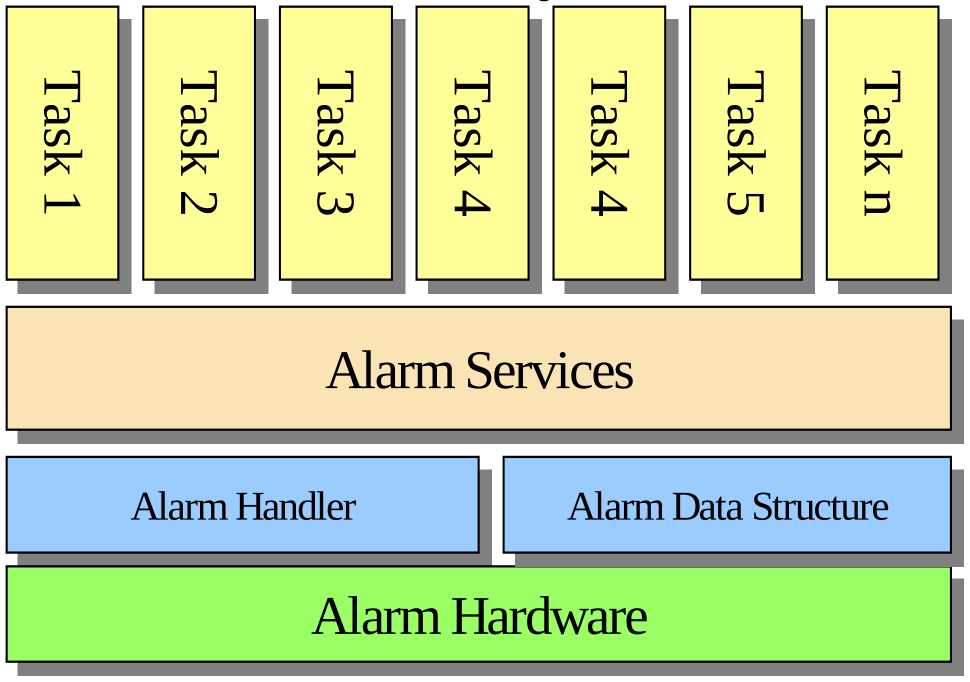
PPSM-GT Alarm services are used for handling the RTC alarm of DragonBall microprocessors. The APIs enable ease of use and support multiple one-shot or periodic alarms. There are 4 main blocks in Alarm services (see Figure 15.1).

- The Hardware alarm block is the RTC alarm register in the RTC module of a DragonBall microprocessor.
- The Alarm data structure block is the memory buffer that stores all the alarm setting information. PPSM-GT will perform automatic housekeeping for alarm usage.
- The Alarm Handler block is the processing center for alarm activity. The functions include:
  - Creation and deletion of alarms
  - Monitoring of alarm status
  - Housekeeping of alarm resources
- The API block interfaces with the application for RTC information. The gathered information is handled by the Alarm handler block.

This chapter is organized into the following main sections:

- Alarm Services Fundamentals
- Programming Using the Alarm Services
- Summary
- Code Example

**Figure 15.1 Alarm Services Block Diagram**



## Alarm Services Fundamentals

PPSM-GT supports a variety of alarm ranging from event alarm that has specific day and time to periodic alarms that expired periodically. In general, there are 2 main types of alarm that differ in functions and usage. They are event and periodic alarms:

### Event Alarm

Event alarms are alarms that are created by AlmCreate() and they are created with date, time, or date and time related. When creating an event alarm there is also a choice to have it as a one shot or repeated alarm type. That is an alarm could be set to expire every year at the same day and time, or every hour at the same time etc. The type selections are shown in Table 15.1.

Event alarms when creating can be deleted by using any one of the alarm deleted APIs such as AlmDelete(), AlmDelAfter(), AlmDelBefore(), and AlmDelAll(). In addition, when an event alarm type is set as one shot alarm type, it will be deleted by the system upon expired. No further alarm deleted APIs is required to delete the alarm in this situation.

**Table 15.1 Event Alarm Type**

| Input Parameter | Alarm Type | Alarm removed |
|---|---|---|
| ALM_ONESHOT | One-shot alarm | On expiring or any alarm delete API |
| ALM_EVERYDAY | Periodic daily alarm | Any alarm delete API |
| ALM_DAYOFWEEK | Periodic specific day of week | Any alarm delete API |
| ALM_DAYOFMONTH | Periodic specific day in a month | Any alarm delete API |
| ALM_ANNIVERSARY | Periodic anniversary day | Any alarm delete API |

## Periodic Alarm

Periodic alarms are alarms that expired when a certain period is up. The periodic could be every hour, minute, second or day. These sort of alarms are day and time specific as when set it expired every day or hour depending on what sort of periodic alarm is set. To set a periodic alarm, the AlmSetPeriodId() is used, and the inputs are as in Table 15.2

**Table 15.2 Type of Sequential Alarms**

| Sequential Alarm Type | Description |
|---|---|
| RTC_PERI_HOUR | For every hour |
| RTC_PERI_MINUTE | For every minute |
| RTC_PERI_SECOND | For every second |
| RTC_PERI_MIDNIGHT | For every day at midnight |

As mentioned if the periodic alarm is set as hourly, then the alarm will expired at the top of every hour and so for regardless of the day and time.

The system also support multiple periodic alarms such that there could be one for every hour, minute and day etc.

To delete a sequential alarm, the AlmSetPeriodId() is again used and the input are as shown in Table 15.3

**Table 15.3 Deleting of Sequential Alarms**

| Sequential Alarm Type | Description |
|---|---|
| RTC_PERI_NO_HOUR | To delete every hour alarm |
| RTC_PERI_NO_MINUTE | To delete every minute alarm |
| RTC_PERI_NO_SECOND | To delete every second alarm |
| RTC_PERI_NO_MIDNIGHT | To delete every day at midnight alarm |
| RTC_PERI_NONE | To delete all sequential alarms |

## Programming Using the Alarm Services

### Creating Alarm

| | |
|---|---|
| STATUS **AlmCreate**(P_U32 alarmId, U16 year, U8 month, U8 day, U8 hour, U8 minute, U8 type) | For creating event alarm only. When a new alarm is created, the system will insert the new alarm into its appropriate location in the alarm list. To have a one shot alarm or periodic type of alarm, the type input is as shown in Table 15.1 |

For example, assume there are two alarms in the alarm list:

1. Set at 8:00 a.m. on 1/3/2000
2. Set at 10:00 a.m. on 1/3/2000.

If a new alarm is set at 9:00 a.m. on 1/3/2000, the new alarm will replace the second alarm, which then becomes the third alarm on the list.

If there is no alarm previously set, then the new alarm is the first alarm in the link list.

PPSM-GT also validates the input for date and time. Regardless of whether the setting is for a yearly, monthly, daily or hourly alarm, all input fields must be valid. Otherwise an error message ERR_ALM_CREATE for invalid data is returned.

If the alarm has been set, the system will return an alarm Id. This Id is required to delete the alarm.

## Deleting Alarm

| | |
|---|---|
| void **AlmDelete**(ALARM_ID alarmId, TASK_ID taskId) | AlmDelete( ) deletes one alarm at a time based on the alarm ID and task ID provided.<br><br>If taskId is not “0,” the system will check the taskId against the alarmId.<br><br>If the taskId or the alarmId is invalid, the system will return the error message ERR_ALM_INVALID_DEL.<br><br>To delete the alarm regardless of the taskId, set the taskId field to “0.” When the taskId is 0, the system will just delete the alarm with the alarmId.<br><br>Note:<br>This API is used to delete event alarm only, to delete periodic alarm, please refer to AlmSetPeriodId(). |
| void **AlmDelBefore**(U16 year, U8 month, U8 day, U8 hour, U8 minute, TASK_ID taskId) | AlmDelAfter( ) deletes alarms that are set before the specified time and the specified task. If taskId is 0, then all alarms that are set before the specified time will be deleted.<br>Note:<br>This API is used to delete event alarm only, to delete periodic alarm, please refer to AlmSetPeriodId(). |

| | |
|---|---|
| void **AlmDelAfter(**U16 year, U8 month, U8 day, U8 hour, U8 minute, TASK_ID taskId) | AlmDelBefore( ) deletes alarms that are set after the specified time and the specified task. If taskId is 0, then all alarms that are set after the specified time will be deleted.<br>Note:<br>This API is used to delete event alarm only, to delete periodic alarm, please refer to AlmSetPeriodId(). |
| void **AlmDeleteAll**(TASK_ID taskId) | AlmDeleteAll( ) deletes all alarms that are set by the specified task. If taskId is 0, then all alarms will be deleted.<br>Note:<br>This API is used to delete event alarm only, to delete periodic alarm, please refer to AlmSetPeriodId(). |

All the alarm-delete APIs have an input field for task ID. This is the task ID of the task that sets the alarms, and it is required to delete the alarm created by that task. For general deletion, the taskId field is “0.” That is, to delete all alarms set before April 1, 2001, at 8:00 p.m., call AlmDelBefore(2001,4,1,8,0,0). Notice that the taskId input field is “0.”

If taskId is not “0,” the system will validate the taskId, and if the task ID is invalid, the system will return the ERR_ALM_INVALID_DEL error message.

PPSM-GT will automatically delete expired one-shot alarms and non-periodic alarms. Periodic alarms—such as at every second, at every minute, hourly, daily, weekly, monthly and yearly—have to be deleted manually.

## Getting Alarm Information

Information on the alarm setting information can be obtained with the AlmGetCurrent( ), AlmGetId( ), AlmGetIdByTime( ), and AlmGetNext( ) APIs.

| | |
|---|---|
| STATUS **AlmGetCurrent**(ALARM_ID alarmId, P_U32 taskId, P_U16 year, P_U8 month, P_U8 day, P_U8 hour, P_U8 minute, P_U8 type) | AlmGetCurrent( ) returns the information of the specified alarm element. |
| ALARM_ID **AlmGetId**(void) | AlmGetId( ) returns the most recent expired alarm ID. |
| STATUS **AlmGetIdByTime(**P_U32 alarmId, U16 year, U8 month, U8 day, U8 hour, U8 minute) | AlmGetIdByTime( ) returns the first alarm that matches the specified time. |
| STATUS **AlmGetNext**(ALARM_ID alarmId, P_U32 taskId, P_U16 year, P_U8 month, P_U8 day, P_U8 hour, P_U8 minute, P_U8 type) | AlmGetNext( ) returns the next alarm's information of the specified alarm element. |

## Creating the Periodic Alarm

| | |
|---|---|
| STATUS **AlmSetPeriodId**(P_U32 alarmId, U8 period) | Set the periodic alarms for every second, minute, hour and day. To delete all periodic alarms set period to none.<br>• RTC_PERI_NONE<br>• RTC_PERI_HOUR<br>• RTC_PERI_MINUTE<br>• RTC_PERI_SECOND<br>• RTC_PERI_MIDNIGHT<br>• RTC_PERI_NO_HOUR<br>• RTC_PERI_NO_MINUTE<br>• RTC_PERI_NO_SECOND<br>• RTC_PERI_NO_MIDNIGHT<br>Refer to Periodic Alarm for more details. |

## Summary

The Alarm services support four type of APIs for creating, deleting, getting, and setting alarm information.

There are 2 main types of alarm that differ in functions and usage. They are event and periodic alarms.

Event alarm could be set as one-shot alarms are alarms that only expire once, or repeated alarms that expire at a fixed periodic interval. Daily, monthly, and yearly alarms are examples of periodic alarms.

Periodic alarms are alarms that once set will expire at fixed period regardless of day and time. For example when a periodic hourly alarm is set, it will expire at the top of every hour regardless of the time of the day and day of the month.

## Code Example

**Listing 15.1 Sample Alarm Services APIs**

```
/* Variable definition*/

U32     AlarmId;
U32     taskAId, taskBId;

void Alarm ( )

{

/* Creating a one shot alarm for 7/14/2000 at 23:00 hour*/
    AlmCreate(&AlarmId, 2000, 7, 14, 23, 0, ALM_ONESHOT);

/* Deleting a alarm that just expired*/

    AlmDelete(AlarmGetId(), 0);

/* Deleting alarms that was created by task A that are set  after 
4/1/2004    */
```

```
    AlmDelAfter(2004/4/1,0,0,taskAId);

/* Deleting all alarms that was created by task B */

    AlmDeleteAll(2004/4/1,0,0,taskBId);

/* Creating a yearly alarm for 2/14/2003 at 08:00 hour */

    AlmCreate(&AlarmId, 2003, 2, 14, 8, 0, ALM_ANNIVERSARY);

/* Creating a hourly periodic alarm*/

    AlmSetPeriodId(&AlarmId, ALM_PERI_HOUR);

/* Deleting the hourly periodic alarm*/

    AlmSetPeriodId(&AlarmId, ALM_PERI_NO_HOUR);
```

# 16 Application Download Services

Application download services enable the system integrator or software developer to provide an area of the system that allows third-party software developers to write their own software and download it into the system for execution.

In PPSM-GT, the downloaded application is referred to as the *application image* to differentiate between the system resident application and the downloaded application. Therefore when reading this chapter it is important to remember that when referring to an application image, it is not any graphic image, but the downloaded application.

PPSM-GT provides APIs such as AppConvertImage(), which allows the downloaded application image to be converted into the download application in the system, and AppDeleteImage(), which deletes the download application and frees the memory allocated for the application.

Developers have to design and allocate the amount of memory needed for download application usage.

This chapter is organized in the following topics:

- "Downloading Application Fundamental,"
- Programming Using Application Download Services

## Downloading Application Fundamental

Applications are the basis that support the functions and features of a product in the PPSM-GT environment. With the additional of downloaded application, there are now two avenues to support the functions and features of a product. System integrators and device developer could provide functions and features through resident applications or allow third parties developer to develop applications through application images.

Both resident applications and download applications are constructed with the system application services and operate similarly in the PPSM-GT environment except those as highlighted in Table 16.1

**Table 16.1 Differences between Resident and Download Application**

| | Resident Application | Download Application |
|---|---|---|
| Description | Built by manufacturer and installed in the system of the product and normally stored in ROM or Flash | Built by third party vendor and are downloaded during runtime. Could be stored in Flash or RAM |
| Type | DEF_APP_TYPE | DNL_APP_TYPE |
| Priority | 1 to 15 | 8 only |

## Download Application Architecture

**Figure 16.1 Download Application Architecture**

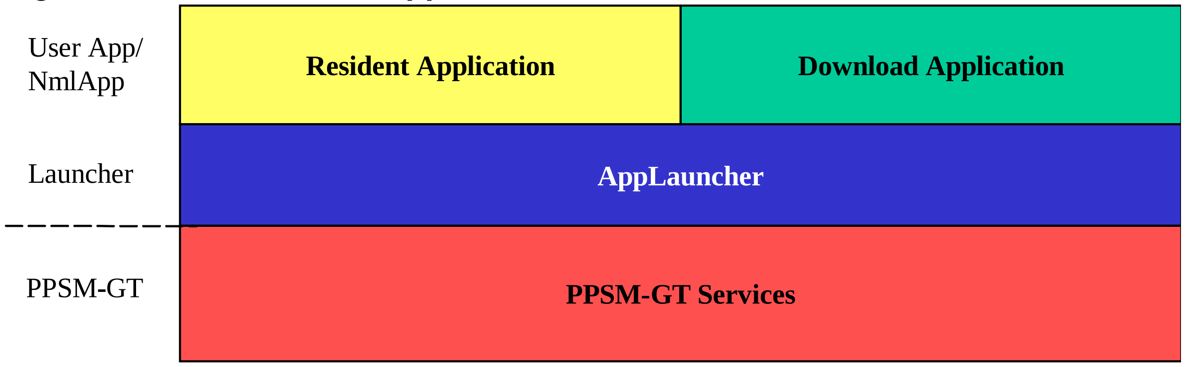


Figure 16.1 shows the system architecture to support a download application. The current PPSM-GT services provide all necessary building blocks to support both resident and download applications. The system application services is used to create the application environment and the kernel and the rest of the services supports the rest of the functions.

In the effort to demonstrate the support of download application, the following is method is provided for reference. This is just one of the many ways available to build a system that handles both resident and download application. Designers are free to follow the same design methodology or used their own design.

The suggested method consist of the following elements:

- Application Launcher
- Application to be downloaded,
- PC program to download the application
- Mechanical to receive download application.
- Mechanical to install the download application to the system
- Mechanism to run the download application.
- Mechanism to remove the download application from the system

## Application Launcher

In most application a main menu task is designed to control the execution of all the applications in the system. The activation of an application could either be by the system or through external stimulation unique to the system design. This concept is especially important in the download application environment to control the execution of the download application. An application launcher is a form of main menu task that supports the execution of the resident applications and the download application. It normally will consist of the following functions and the structure is shown in Table 16.2:

- Initialization
- Task swapping
- Display of the launcher screen
- Handling of inputs, and
- Handling of application images

The first four functions are covered in detail in other chapters and will not be covered in this chapter. For more information please refer to

- Chapter 11, System Application Services for on application initialization and task swapping,
- Chapter 21, Graphic Manipulation Services for display of launcher screen, and
- Chapter 24, Pen Input Handling Services for handling of inputs.

**Table 16.2 Apllication Launcher Structure**

| Application Launcher function | Description |
|---|---|
| SysApp | A normal application structure, a launcher "class" is "Inherited" from SysApp |
| swap to task | The PPSM-GT Task will be swap to |
| swap to app | The PPSM-GT App will be switch to |
| DoEnterApp | member function to do some init job before swap to a SysApp from launcher app |
| DoExitApp | member function to do some clean job before swap to launcher app from SysApp |
| DoInitClientArea | member function to init the area will be used by nmlApp |
| DoDrawMainMeun | member function to draw the main screen of launcher |
| Main screen IC id | The IC id of the main screen |
| BackToMain IC id | The IC id of the ?ack to main?button |
| BackToMain AA id | The AA id of the ?ack to main?button |
| pNmlAppList | a link list of SysApp register to the launcher |
| nmlAppNum | total number of SysApp register to the launcher |
| state | indicate the state of launcher |

The following will cover the topic on handling application images.

### Application to be downloaded

The application to be download or application images must follow the predefined structure in order that the image can operate properly in PPSM-GT environment. Table 16.3 describes the application image structure as predefined by PPSM-GT.

PPSM-GT system will check and understand the download application imge before deciding how to handle it. It the predefined structure is not adhere to, PPSM-GT will ignore the download image.

**Table 16.3 Application Image Structure**

| |
|---|
| Package name : Application name |
| Package type : PKG_DNL_APP_TYPE |
| PackageHeader.s version : Distinguish from different release |
| Writer of this DnlApp : Name of the writer |
| Product ID : Assigned by writer |
| Version number of this product |
| Initialized data address |
| Initialized data size |
| Un-initialized data size |
| Entry point of the DnlApp |
| Task stack size |
| Number of bit per pixel using |
| Large application icon width |
| Large application icon height |
| Large application icon bitmap |
| Entry callback function |
| Exit callback function |

There will be a main task for the application that can create other tasks. This main task has a fixed priority level that is designed by the integrator. In the image, the main task's stack size will be specified. The main task of the application will be created and activated automatically when AppConvertImage() is called. The integrator will call AppSwitch() to switch to this downloaded application.

### PC program to download the application

The download process is initiated by a PC download program.

PPSM-GT utility program provide BBUG program.

The BBUG project is written by VC++ 6.0, WinNT. It is not related to bootstrap, only the CSerial class is used to control the COM port of PC.

Please refer to the application note for "How to create down load application".

### Mechanical to receive download application.

A Launcher tasks is a special task created to receive the download application. It is created in the application launcher to handle the launching of a application image. A launcher task behaves like a communication task that, when executed, waits for the input of the download application. Developers can choose to accept the download application through UART, IrDA or another communication method. Developers have to design the appropriate driver for the download application. A launcher task will be needed to:

- download the application image into RAM
- optionally burn the application image to flash
- call AppConvertImage() to create the task and application
- send an event to the menu task to register the new application

### Mechanical to install the download application to the system

The download application or known as application image in the PPSM-GT environment is installed to the system by calling AppConvertImage(). This function will allocate memory for the RAM and data and then create the main task. Inside the main task, it should create other tasks as required and create the graphic context for the application. .

### Mechanism to run the download application.

The AppConvertImage() API interprets the downloaded image and creates the application and the first task in the download application. It will not switch the application. The system integrator can use AppSwitch() to start the application.

### Mechanism to remove the download application from the system

The AppDeleteImage() API will remove the application image from the system. The memory area for the image will be freed by the task creating it but not by this API.

## Client area and System area

There is no restriction on the client and system areas in the PPSM-GT environment, however, system integrators and device developers should consider introducing some form of restrictions when supporting application images on their devices. Restriction such as alloting only certain portion of the memory that could only be used by third parties application developers and also introducing partition of the LCD can be used by application images. Such practices would go a long way to ensure that the device performance would not be compromised by improper execution of 3rd party application images.

### Trap call

There are 8 level of TRAP call available in dragonball processor and TRAP 1 is reserved and used by the PPSM-GT system. The rest of the TRAP call are open and available to system integrators and device designer to use to support application images that need to call some addition function for initialization and get the system information.

## Programming Using Application Download Services

### Converting downloaded application image

| | |
|---|---|
| STATUS **AppConvertImage**(U32 imagePtr, P_APP_ID pAppId, P_TASK_ID pTaskId) | This function will allocate memory for the RAM and data and then create the main task. Inside the main task, it should create other tasks as required and create the graphic context for the application. The system integrator can use AppSwitch() to start this application.<br><br>The API interprets the downloaded image and creates the application and the first task in the download application. It will not switch the application until the user calls AppSwitch(). However, the created tasks in the download application will start immediately. |

## Deleting download application image

| | |
|---|---|
| STATUS **AppDeleteImage**(APP_ID pAppId) | It deletes the download application and all tasks created for this application.<br><br>The memory area for the image will be freed by the task creating it but not by this API. |

## Creating task in download application

| | |
|---|---|
| STATUS **AppCreateTask**(P_TASK_ID pTaskId, APP_ID appId, P_VOID pFunc, U32 stackSize) | It creates a task for the downloaded application. However, the main task in the download application cannot be created using this API. |

## Deleting task in download application

| | |
|---|---|
| STATUS **AppDeleteTask**(APP_ID appId, TASK_ID taskId) | It deletes the task created in the download application. |

# Code Examples

**Listing 16.1 Header file of DnlApp**

```
_PKG_TYPE EQU $1 ; This package is Download application
_PKG_PDT_ID EQU $00000001 ; Product ID assigned by Software house
_PKG_PDT_VER EQU $01000001 ; Version number of the product
assigned by Software house
_PKG_DNL_APP_STACK EQU $A000 ; Task stack size
_PKG_DNL_APP_BPP EQU $2 ;1 or 2 or 4bpp
_PKG_DNL_APP_ICO_W EQU $30; App. icon width and height: 48x48
_PKG_DNL_APP_ICO_H EQU $30

SECTION .header
```

```
START
_pPackageName DC.L _gpPackageName ;0 Package name
_PackageType DC.W _PKG_TYPE ;1 Package type
_ThisHeaderVersion DC.W _PKG_HDR_VER_1_1_0 ;1 PackageHeader.s
version
_pPkgManufacturerName DC.L _gpPkgManufacturerName;2 Writer of
this DnlApp
_PkgProductId DC.L _PKG_PDT_ID ;3 Product ID assigned by writer
_PkgProductVersion DC.L _PKG_PDT_VER ;4 Version number of this
product
_pAppDataStart DC.L _DATASTART ;5 Initialized data address
_AppDataSize DC.L _DATASIZE ;6 Initialized data size
_AppRamSize DC.L _RAMSIZE ;7 Un-initialized data size
_pAppTaskStart DC.L _gPpsmgtApp ;8 Entry point of the DnlApp
_AppTaskStackSize DC.L _PKG_DNL_APP_STACK ;9 Task stack size
_AppBitDepth DC.W _PKG_DNL_APP_BPP ;0xA Number of bit per pixel
using
_AppIconWidth DC.B _PKG_DNL_APP_ICO_W ;0xA Large application icon
width
_AppIconHeight DC.B _PKG_DNL_APP_ICO_H ;0xA Large application
icon height
_pAppIconImage DC.L _gpAppIconImage ;0xB Large application icon
bitmap
_pAppEntryCallback DC.L _gAppEntryCallback ;0xC Entry callback
function
_pAppExitCallback DC.L _gAppExitCallback ;0xD Exit callback
function

source file of application:
/* The application task */

void gPpsmgtApp(void)
{
P_EVENT pEvent;
while (1)
{
```

# 17 Audio Management Services

PPSM-GT supports three types of audio: tone, wave and melody playing. 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 Modulator) module is in use by another task or application.
- An event will be sent to the task that called Audio Services to indicate that the audio playing has finished.

This chapter is organized into the following main sections:

## Audio Management Services Fundamentals

### Tone service

The AudPlayTone( ) API plays a sound with a specified frequency (tone) for a specified length of time. Apart from these characteristics, the pointers to the tone frequency file and the file size are needed. Examples of use are to generate a dial tone or a laser-like sound for game playing.

## Wave service

The AudPlayWave( ) API plays back an audio wave file in PCM (Pulse Code Modulation) format that can be generated by many audio programs. The general usage of this service is to play back a piece of audio, such as a segment of speech, that is recorded by a computer or any audio device.

Two basic characteristics, sampling rate and sample bit resolution, characterize this kind of audio service. A common example of such service is an audio CD (compact disc), which provides 44.1 kHz, 16-bit, stereo audio quality. In contrast, the API service only provides mono service and 8-bit resolution. Apart from these characteristics, a pointer to the wave data file and file size is needed.

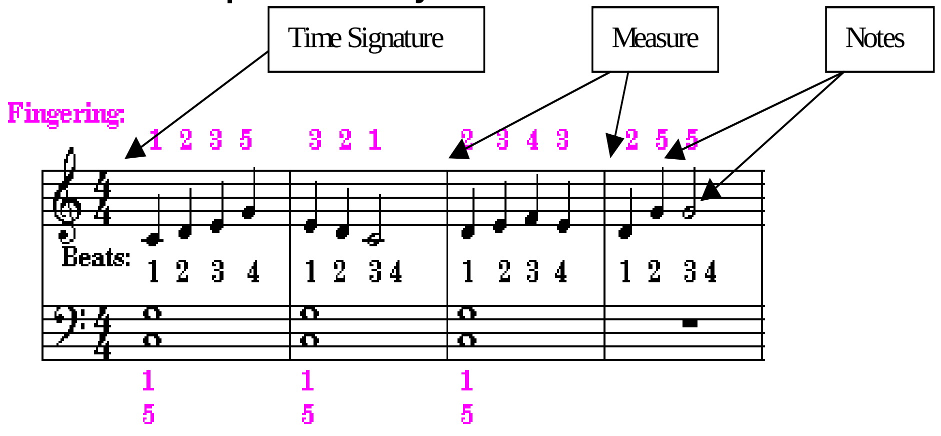
## Melody service

The AudPlayMelody( ) API plays back melody notes like piano playing. Each note has a specific pitch, and each pitch has a particular frequency associated with it. Including sharps and flats, there are 12 actual notes in an octave.

The most common octave has a base frequency of 220 Hz. The next higher octave has a base frequency of 440 Hz, and so on.

Figure 17.1 shows an example of melody notes.

**Figure 17.1 Example of melody notes**



## PPSM Music File Format

The audio service allows users to play a melody by using the PPSM-GT Music File Format (PMF). The PMF file is simple and easily used on the system.

The header identifies all the important parameters, and the melody data is stored in a simple dump. Table 17.1 shows the format of the PFM header.

If the same note is going to be played twice in succession, a pause signal will be inserted between the notes automatically. The duration of the pause signal is programmable and fixed during compilation.

**Table 17.1 PMF Header Description**

| Bytes | Type | Description |
|---|---|---|
| 1 | Version number | It is data to indicate the version number of the PMF file. |
| 1 | The tempo of the melody | The rate at which a measure is played. There are three types of tempo. They are ALLEGRO, MODERATO and LARGHETTO.<br>01 = ALLEGRO means playing 4 quarter notes per second.<br>02 = MODERATO means playing 2 quarter notes per second.<br>03 = LARGHETTO means playing 1 quarter note per second. |
| 2 | Total number of notes in the file | The total number of the “measure” to form the PMF file. |
| n | Name of the melody, terminated by a “0x0000” character | It is a string, which is used to record the name of the PMF file. |
| P_NOTE | Note | A pointer to the note data structure. |

**Table 17.2 PPSM-GT Note Description**

| Notes | | |
|---|---|---|
| NOTE_SILENCE | NOTE_G4_FLAT | NOTE_D6_SHARP |
| | NOTE_G4 | NOTE_E6_FLAT |
| NOTE_C3 | NOTE_G4_SHARP | NOTE_E6 |
| NOTE_C3_SHARP | NOTE_A4_FLAT | NOTE_F6 |
| NOTE_D3_FLAT | NOTE_A4 | NOTE_F6_SHARP |
| NOTE_D3 | NOTE_A4_SHARP | NOTE_G6_FLAT |
| NOTE_D3_SHARP | NOTE_B4_FLAT | NOTE_G6 |
| NOTE_E3_FLAT | NOTE_B4 | NOTE_G6_SHARP |
| NOTE_E3 | NOTE_C5 | NOTE_A6_FLAT |
| NOTE_F3 | NOTE_C5_SHARP | NOTE_A6 |
| NOTE_F3_SHARP | NOTE_D5_FLAT | NOTE_A6_SHARP |
| NOTE_G3_FLAT | NOTE_D5 | NOTE_B6_FLAT |
| NOTE_G3 | NOTE_D5_SHARP | NOTE_B6 |
| NOTE_G3_SHARP | NOTE_E5_FLAT | NOTE_C7 |
| NOTE_A3_FLAT | NOTE_E5 | NOTE_C7_SHARP |
| NOTE_A3 | NOTE_F5 | NOTE_D7_FLAT |
| NOTE_A3_SHARP | NOTE_F5_SHARP | NOTE_D7 |
| NOTE_B3_FLAT | NOTE_G5_FLAT | NOTE_D7_SHARP |
| NOTE_B3 | NOTE_G5 | NOTE_E7_FLAT |
| NOTE_C4 | NOTE_G5_SHARP | NOTE_E7 |
| NOTE_C4_SHARP | NOTE_A5_FLAT | NOTE_F7 |
| NOTE_D4_FLAT | NOTE_A5 | NOTE_F7_SHARP |
| NOTE_D4 | NOTE_A5_SHARP | NOTE_G7_FLAT |
| NOTE_D4_SHARP | NOTE_B5_FLAT | NOTE_G7 |
| NOTE_E4_FLAT | NOTE_B5 | NOTE_G7_SHARP |
| NOTE_E4 | NOTE_C6 | NOTE_A7_FLAT |
| NOTE_F4 | NOTE_C6_SHARP | NOTE_A7 |
| NOTE_F4_SHARP | NOTE_D6_FLAT | NOTE_A7_SHARP |
| | NOTE_D6 | NOTE_B7_FLAT |
| | | NOTE_B7 |

**Table 17.3 PPSM-GT Pitch Description**

| Note Type | PPSM-GT's notation of PITCH LENGTH |
|---|---|
| Whole Note | NOTE_WHOLE |
| Half Note | NOTE_HALF |
| Quarter Note | NOTE_QUARTER |

| Note Type | PPSM-GT's notation of PITCH LENGTH |
|---|---|
| Eighth Note | NOTE_EIGHTH |
| Sixteenth Note | NOTE_SIXTEENTH |
| Thirtieth Note | NOTE_THIRTIETH (1/32 * NOTE_WHOLE) |
| Sixtieth Note | NOTE_SIXTIETH (1/64 * NOTE_WHOLE) |

### Writing up a PMF file

The write up of the PMF file is divided into the following steps:

1. Create the PMF header information as in Table 17.1.
2. Code the music source using the PPSM-GT notation, as shown in Table 17.2 and Table 17.3.
3. Include the file as a *.h file.

## Programming Using Audio Management Services

### Playing Tone

| | |
|---|---|
| STATUS **AudPlayTone**(P_U16 toneData, U32 toneSize, U16 toneDuration, U8 autoRepeat) See Table 17.4 for description | PPSM-GT supports tone playing through the PWM module. Tone playing can play a melody with a user-specified fixed duration and changeable frequencies throughout that duration. For better frequency resolution, the tone frequency is limited to between 31 Hz and 4048 Hz. |

**Table 17.4 Tone Input Description**

| Input Field | Description |
|---|---|
| toneData | The pointer to the tone sequence, with frequencies between 31 Hz and 4048 Hz. |
| toneSize | Total number of tone frequencies to be played. |
| toneDuration | The duration of each tone frequency.<br><br>For DragonBall EZ:<br>TONE_DUR_512Hz<br>TONE_DUR_256Hz<br>TONE_DUR_128Hz<br>TONE_DUR_64Hz<br>TONE_DUR_32Hz<br>TONE_DUR_16Hz<br>TONE_DUR_8Hz<br>TONE_DUR_4Hz<br><br>For original DragonBall:<br>0 to 1000 (length of duration in number of milliseconds) |
| autoRepeat | To indicate if auto-repeat is needed or not.<br>0 - no auto-repeat<br>1 - auto-repeat |

## Setting up for tone music

| | |
|---|---|
| STATUS **AudSetTone**(void) | It sets the PWM register to the specified frequency from 4.048 kHz to 31.625 Hz and starts to play tone data one by one. This API can be used for playing the same tone after it has been stopped. |

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** It is impossible to play a tone whose value of frequency is less than the value of duration, since the duration of this frequency is longer than the allowed duration.

## Playing Wave Format

PPSM-GT audio services can play back a PCM (Pulse Code Modulation) audio wave file that can be generated by many audio programs. Two PPSM-GT audio tools can play waves: AdvAudioPlayWave( ) and AudioPlayWave( ).

| | |
|---|---|
| STATUS **AudPlayWave**(P_U8 waveData, U32 waveSize, U8 samplingRate) See Table 17.5 for description | To play wave music. |

| | |
|---|---|
| STATUS **AudAdvPlayWave**(P_U8 waveData, U32 waveSize, U8 prescaler, U8 repeat, U8 clksel) | AdvAudPlayWave( ) is provided for users with solid knowledge of PWM who want to have advanced configuration control over the PWM module. |

Table 17.5 **Wave Input Description**

| Wave Inputs | 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 |
| *prescaler (see DragonBall EZ user's manual)* | Bits 14–8 of the PWM control register; value from 0 to 127 |
| *repeat (see DragonBall EZ user's manual)* | Bits 2 and 3 of the PWM control register; value from 0 to 3 |
| *clksel (see DragonBall EZ user's manual)* | Bits 0 and 1 of the PWM control register; value from 0 to 3 |

The sampling rate can be calculated with the input parameters identified in Table 17.5:

$$SamplingRate = ((16.58MHz)/(prescalar+1)/(clksel)/(repeat)/256$$

For more detailed information, please refer to the DragonBall EZ and DragonBall VZ user's manuals. (The PWM control register on the DragonBall EZ is the PWM 1 control register on the DragonBall VZ.)

## Setting up the Audio Melody

| | |
|---|---|
| STATUS **AudSetMelody**(P_PMF pmf) | It is a function to set up the audio melody. |

## Playing Melody Music

| | |
|---|---|
| STATUS **AudPlayMelody**(void) | Start melody playing according to pre-defined setting |

## Stopping the audio playing

An audio stops after it has finished or when the user has called AudStopTone( ), AudStopWave( ), or AudStopMelody( ) to stop the playing. After audio playing stops, an interrupt is sent to the task that called audio services to indicate that the audio playing is finished.

| | |
|---|---|
| STATUS **AudStopTone**(void) | It stops playing the tone data. |
| STATUS **AudStopWave**(void) | It stops playing the wave data. |
| STATUS **AudStopMelody**(void) | It stops playing the melody data. |

## Pausing the audio melody playing

| | |
|---|---|
| STATUS **AudPauseMelody**(void) | Pausing melody playing is allowed in PPSM-GT. When the AudPauseMelody( ) API is used, the system stops the playing of the melody notes and remembers where it stopped. It will resume playing when AudPlayMelody( ) is used.<br>AudPauseMelody( ), when called, pauses the melody data that is playing. |

## Inquiring about the playcounter

| | |
|---|---|
| STATUS **AudGetCount**(P_U16 playcounter) | To return the playcounter for indicating the number of notes played |

## Inquiring about name of melody in PMF file

| | |
|---|---|
| STATUS **AudGetName**(P_U16 pmfname) | To return the PMF name |

### Inquiring about pitch length of melody

| | |
|---|---|
| U16 **AudGetNoteLength**(void) | To return the pitch length |

### Inquiring about number of notes in melody

| | |
|---|---|
| STATUS **AudGetNumofNote**(P_U16 sumofnote) | To return the number of notes |

### Inquiring about audio tool status

| | |
|---|---|
| U8 **AudGetStatus**(void) | It returns the audio tool status to the application. |

### Inquiring about tone duration

| | |
|---|---|
| U16 **AudGetToneDur**(void) | To return the tone duration |

## Summary

PPSM-GT audio management services can be used to generate tone, wave and melody music. The audio management services are set; forget APIs that set the tone, wave and melody musical notes, and the system will generate the music.

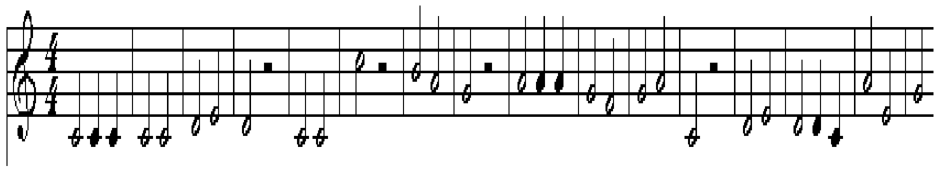
The melody music services use a PMF format to play the music. It has a header and data section. The description of the PMF header appears in Table 17.1.

## Code Examples

The following examples illustrate the creation of the PMF file and how to use the audio management services to generate tone, wave and melody music. These examples are provided for reference and, if modified, could be used in an actual application.

## Creating a PMF file

Consider the following musical score, the theme from “My Girl.”



The PMF header information is as follows:

- 01 01; 01 means version number 0.1 and ALLEGRO means playing 4 quarter notes per second.
- 00 21; 00 21 means that there are 33 notes in the song in total.
- 47 49 52 4C; 47,49,52 and 4C mean the ASCII characters “GIRL,” which is the name of the song.
- 00 00; 00 00 signifies the terminator of the header information.

**Listing 17.1 The PMF file example**

```
const U16  pmf2[] = {
0x0101,0x0021,0x4749,0x524C,
0x0000,
NOTE_C3,        NOTE_HALF,
NOTE_C3,        NOTE_QUARTER,
NOTE_C3,        NOTE_QUARTER,
NOTE_C3,        NOTE_HALF,
NOTE_C3,        NOTE_HALF,
NOTE_D3,        NOTE_HALF,
NOTE_E3,        NOTE_HALF,
NOTE_D3,        NOTE_HALF,
NOTE_SILENCE,   NOTE_HALF,
NOTE_C3,        NOTE_HALF,
NOTE_C3,        NOTE_HALF,
NOTE_C4,        NOTE_HALF,
NOTE_SILENCE,   NOTE_HALF,
NOTE_B3,        NOTE_HALF,
NOTE_A3,        NOTE_HALF,
NOTE_G3,        NOTE_HALF,
NOTE_A3,        NOTE_HALF,
NOTE_A3,        NOTE_QUARTER,
NOTE_A3,        NOTE_QUARTER,
NOTE_G3,        NOTE_HALF,
NOTE_F3,        NOTE_HALF,
NOTE_G3,        NOTE_HALF,
NOTE_A3,        NOTE_HALF,
NOTE_C3,        NOTE_HALF,
NOTE_SILENCE,   NOTE_HALF,
NOTE_D3,        NOTE_HALF,
NOTE_E3,        NOTE_HALF,
NOTE_D3,        NOTE_HALF,
NOTE_D3,        NOTE_QUARTER,
NOTE_C3,        NOTE_QUARTER,
NOTE_A3,        NOTE_HALF,
NOTE_E3,        NOTE_HALF,
NOTE_G3,        NOTE_HALF,
};
```

**Listing 17.2 PPSM-GT tone playing**

```
/*  100Hz, 1000Hz, 500Hz and 600Hz */
U16toneData[] = {100, 1000, 500, 600};

/*  Play a melody with 4 different tone frequencies, each with
250 ms duration */

  AudPlayTone((P_U16)toneData, 4, TONE_DUR_4HZ, 1);
```

**Listing 17.3 PPSM-GT wave playing**

```
/*  Some PWM wave data */
U16waveData[] = {...};

/*  Play a melody with 1000 data bytes at 16 kHz sampling/
reconstruction rate */

    AudioPlayWave((P_U8)waveData, 1000, SAMPLING_16KHZ);
```

**Listing 17.4 PPSM-GT wave playing**

```
/*  Some PWM wave data */
U16waveData[] = {...};
```

```
/*  Play a melody with 1000 data bytes at 16 kHz sampling/
reconstruction rate */
/*  16 kHz = 16.58 Mz/(2 x 1 x 2 x 256) */
    AudAdvPlayWave((P_U8)waveData, 1000, 0, 2, 0);
```

**Listing 17.5 Stopping an audio play**

```
/*  Some PWM wave data */
U16waveData[] = {...};

/*  Play a melody with 1000 data bytes at 16 kHz sampling/
reconstruction rate */
AudioPlayWave((P_U8)waveData, 1000, SAMPLING_16KHZ);

 if (areaId == StopIcon)
  /*  Click icon to stop the wave playing */
  rv = AudioStopWave();
  ....
  break;
}
```

**NOTE** Listing 17.6 provides pseudo-code for using the Audio Management Services for playing a melody.

**Listing 17.6 Playing Melody music**

```
const U16  pmf1[] = {
0x0101,0x003C,0x656d,0x6500,0x0000,
NOTE_F5_SHARP,NOTE_QUARTER,
NOTE_F5_SHARP,NOTE_QUARTER,
NOTE_F5_SHARP,NOTE_QUARTER,
NOTE_D5,NOTE_QUARTER,
NOTE_SILENCE,NOTE_QUARTER,
NOTE_B4,NOTE_QUARTER,
NOTE_SILENCE,NOTE_QUARTER,
NOTE_E5,NOTE_QUARTER,
NOTE_SILENCE,NOTE_QUARTER,
NOTE_E5,NOTE_QUARTER,
NOTE_SILENCE,NOTE_QUARTER,
NOTE_E5,NOTE_QUARTER,
NOTE_G5_SHARP,NOTE_QUARTER,
NOTE_G5_SHARP,NOTE_QUARTER,
NOTE_A5,NOTE_QUARTER,
NOTE_B5,NOTE_QUARTER,
NOTE_A5,NOTE_QUARTER,
NOTE_A5,NOTE_QUARTER,
NOTE_A5,NOTE_QUARTER,
NOTE_E5,NOTE_QUARTER,
NOTE_SILENCE,NOTE_QUARTER,
NOTE_D5,NOTE_QUARTER,
NOTE_SILENCE,NOTE_QUARTER,
NOTE_F5_SHARP,NOTE_QUARTER,
NOTE_SILENCE,NOTE_QUARTER,
NOTE_F5_SHARP,NOTE_QUARTER,
NOTE_SILENCE,NOTE_QUARTER,
NOTE_F5_SHARP,NOTE_QUARTER,
NOTE_SILENCE,NOTE_QUARTER,
NOTE_E5,NOTE_QUARTER,
NOTE_E5,NOTE_QUARTER,
NOTE_F5_SHARP,NOTE_QUARTER,
NOTE_E5,NOTE_QUARTER,
NOTE_F5_SHARP,NOTE_QUARTER,
NOTE_F5_SHARP,NOTE_QUARTER,
NOTE_F5_SHARP,NOTE_QUARTER,
NOTE_D5,NOTE_QUARTER,
NOTE_SILENCE,NOTE_QUARTER,
NOTE_B4,NOTE_QUARTER,
NOTE_SILENCE,NOTE_QUARTER,
NOTE_E5,NOTE_QUARTER,
NOTE_SILENCE,NOTE_QUARTER,
NOTE_E5,NOTE_QUARTER,
NOTE_SILENCE,NOTE_QUARTER,
NOTE_E5,NOTE_QUARTER,
NOTE_G4_SHARP,NOTE_QUARTER,
NOTE_G5_SHARP,NOTE_QUARTER,
NOTE_A5,NOTE_QUARTER,
NOTE_B5,NOTE_QUARTER,
NOTE_A5,NOTE_QUARTER,
NOTE_A5,NOTE_QUARTER,
NOTE_A5,NOTE_QUARTER,
NOTE_E5,NOTE_QUARTER,
NOTE_SILENCE,NOTE_QUARTER,
NOTE_F5_SHARP,NOTE_QUARTER,
NOTE_SILENCE,NOTE_QUARTER,
NOTE_F5_SHARP,NOTE_QUARTER,
NOTE_SILENCE,NOTE_QUARTER,
NOTE_F5_SHARP,NOTE_QUARTER,
NOTE_E5,NOTE_QUARTER,
NOTE_E5,NOTE_QUARTER,
};

/*  Setting up the the melody */

  AudSetMelody(pmf1);


/*   Playing the melody        */
  AudPlayMelody();

/*   Pause the melody if pause action detected    */
    .
    .
    .
    AudPauseMelody();

/*   Playing the melody again if play action detected   */
    .
    .
    .
  AudPlayMelody();

/*   Stopping the melody if stop action detected      */
    .
    .
    .
  AudStopMelody();
```

# 18

# Serial Communication Interface Services

PPSM-GT supports multiple serial communications through the serial communication interfaces (SCIs) in both normal mode and IrDA mode. The exact number of SCI resources supported is hardware limited. (Refer to the appropriate hardware manual for details.) Each resource has an identifier that distinguishes it from others when the common set of SCI services is used to send and receive data through the SCIs.

Each SCI resource also has an internal receive buffer. The default size is specified in the include header file. This size can be dynamically changed during run time with SciSetRxBufSize( ).

By default, SCI resources are disabled and should be enabled before use.

This chapter is organized into the following main sections:

- SCI Services Fundamentals
- Programming Using SCI Services
- Summary
- Code Examples

## SCI Services Fundamentals

The SCI services provide easy-to-use APIs for tasks to enable, configure, and send and receive data serially with or without hardware flow control.

The SCI resources in the PPSM-GT environment are divided into hardware resources and software resources. The hardware resources refer to the physical SCI modules, which consist of the communication UART port available on the hardware platform. The software resources consist of the system memory used for storing the baud rate, parity and other SCI-related information.

## SCI Ownership and Usage

The task that creates the SCI port owns the SCI port. It is the target task of that port; that is, the system will send all SCI-related events to this task. In normal usage, the task that creates the SCI port binds and uses it. When it is done using the SCI port, the task unbinds the SCI port and releases it for other tasks to use. Using the SCI port in this way is a good practice because such SCI usage is straightforward. It is based on the following principle: one task for one SCI port at any one time.

In a multitasking environment, there may be cases when multiple tasks have to use the same SCI port to send information. For such cases, the following points must be considered:

1. The task that creates the SCI port is the target task of that port; all SCI related event will send to this task. In other words, if there are 3 tasks—tasks A, B and C—and task A creates and binds the SCI port, then task A is the target task. If task B or C wants to use the SCI port, the system will send the system events to task A even if those events are for task B or C. Therefore, when using the SCI port in this situation, please keep track of the tasks that are using the SCI ports and redirect the system events to the appropriate tasks.
2. All other tasks can call SCI APIs to control that SCI port as long as the tasks know the SCI portId. It is the programmer's responsibility to exercise proper control of the way the tasks use the SCI ports. Improper control will lead to data corruption due to different tasks sending different information on the same SCI port.
3. All tasks can unbind and close the SCI port if the SCI portId is available. It is again the programmer's responsibility to ensure that no other task is using the SCI port before a task unbinds and closes the SCI port.

## Using the SCI Services

There are two ways to use the SCI services:

1. The three steps method
2. The five steps method

### The three steps method

The three steps method is for applications that just want to send and receive information through the SCI resources for a short period of time. There is no need for saving the SCI configuration for future usage.

**Figure 18.1 The Three Steps Method**

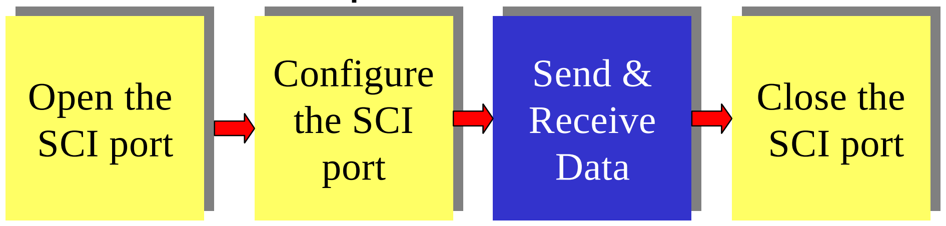


Figure 18.1 and Table 18.1 shows the three steps method. It is straightforward and easy to use. To start using the SCI resources, remember the 3 simple steps.

**Table 18.1 Three Steps Method Description**

| Step | Description |
|---|---|
| Step 1: Open the SCI port with SciOpen() | The system will check whether any SCI hardware resource is available, and return with the SCI resource or an error message if no SCI resources is available. |
| Step 2: Configure the SCI port with SciConfig(). | The SCI resource that is reserved is not configured. SciConfig() will set the baud rate, parity settings, stop bit settings, character length settings, and data transmission time-out settings. |
| Step 3: Close the SCI port with SciClose() | When done using the SCI resource, SciClose() will close and release the SCI resource. |

### The five steps method

The five steps method is an expanded version of the three steps method. It breaks down the three steps into five steps to allow applications that want to send and receive information through the SCI resources and need to re-use the same SCI configuration later.

**Figure 18.2 The Five Steps Method**

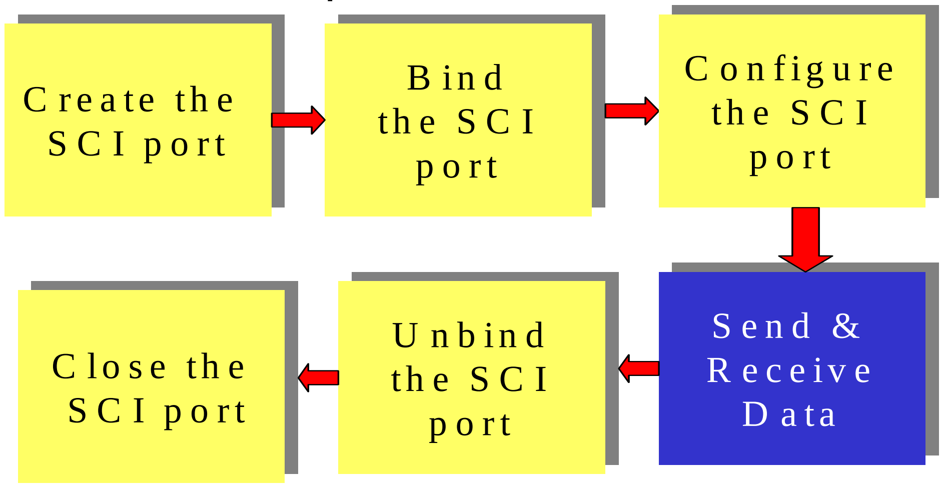


Figure 18.2 shows the five steps method. The five steps method requires the creation of the software ports and the configuration of the hardware ports to occur separately. The software port could be created at any time, but the configuration of the hardware port is under system control. The system is flexible on the usage of the SCI hardware resources, since two or more tasks are able to access the same hardware resource at any one time provided the SCI port is not busy.

The SciBindPort( ) API will check the availability of the SCI resources. Once the hardware port is available, PPSM-GT binds the software port to hardware port, and the task is able to send and receive information through the SCI resources.

When the task has finished SCI operation, there is a choice to release only the hardware resources or both the hardware and software resources. It is good practice to release the SCI port whenever the task has finished using it. PPSM-GT does not force the release of an unused hardware SCI port. Therefore, if any of the tasks hold on to the SCI hardware port without releasing it, the SCI port is bound to the task until the task is terminated.

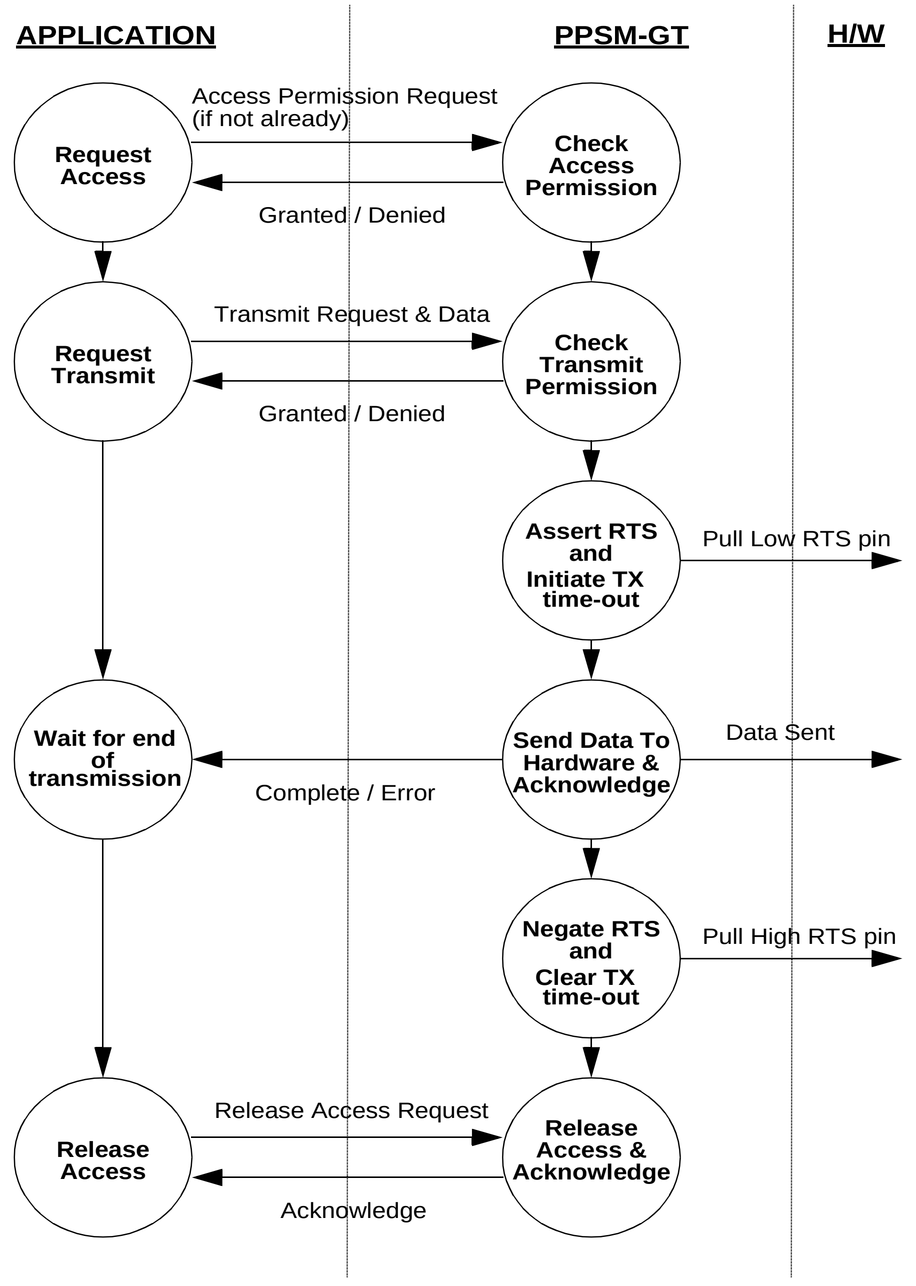
To release only the SCI hardware port, use SciUnbindPort( ). To release both hardware and software ports, use SciClose(). It is not necessary to release the software SCI port every time if the SCI setting is used frequently. It saves the time of reconfiguring the SCI again.

## SCI resources hardware flow control

In PPSM-GT, data communication between the system and other communication devices using the SCI supports RTS, CTS hardware flow control. While hardware flow control is enabled, RTS is asserted automatically when calling SciSend( ) and SciReceive( ). In a null modem configuration, when the PPSM-GT system is the sender, the receiver needs to acknowledge that it is ready to receive by asserting its RTS pin.

Figure 18.3 shows the data transmission with hardware flow control.

**Figure 18.3 SCI Communication Architecture—Data Transmit with RTS, CTS flow control**



When the PPSM-GT system is the receiver, it acknowledges the sender side by asserting the RTS pin. Thus, if the RTS pins of both the PPSM-GT system and the other communication device are asserted, the data transfer can be full-duplex.

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

The API SciSendAbort( ) is used for aborting the transmission. Also, this API can return the current position of the software send buffer and the number of bytes that have been transmitted by the PPSM-GT system with appropriate input flag.

## SCI Configurations

PPSM-GT allows applications to configure the SCI to operate at various baud rates, parity settings, stop bit settings, character length settings, and data transmission time-out settings.

There is no default SCI configuration after creating a software SCI port. Please ensure that the SCI is configured properly by calling SciConfig( ) before initiating any data communication.

### Baud rate supported

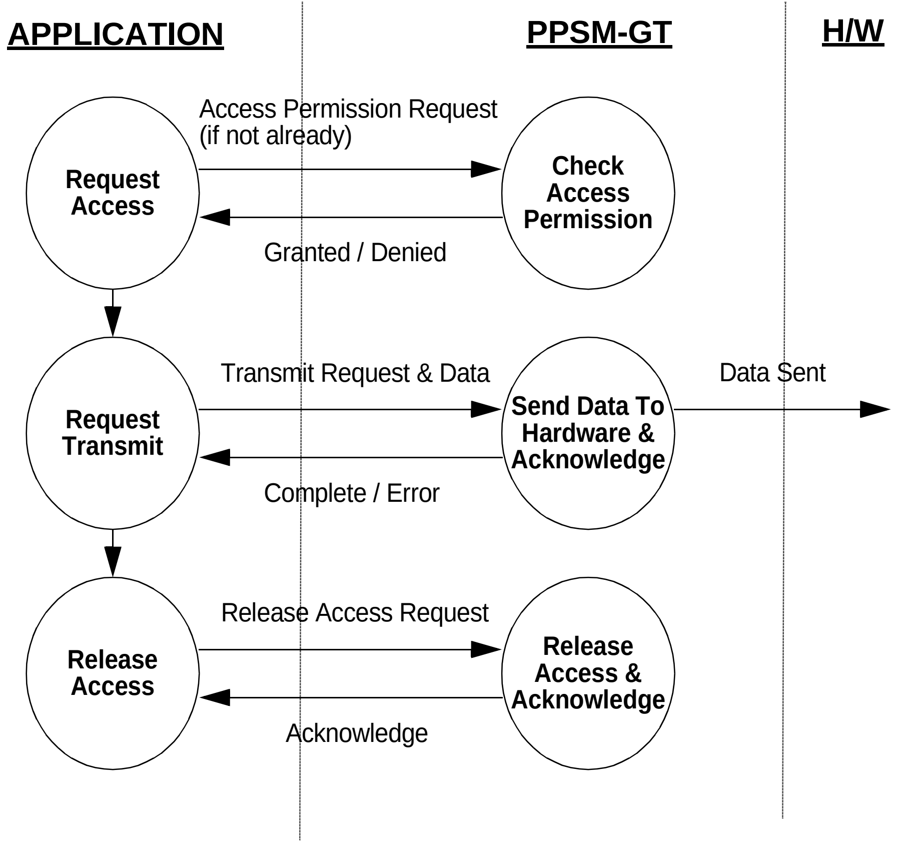
PPSM-GT can support a baud rate from 600 bps to 115,200 bps. Please note that the baud rate supported is also hardware limited, and some hardware is not able to run at the top speed.

## Initiating a Send Request

Figure 18.4 shows the data transmission using the SCI communication architecture.

**Figure 18.4 SCI Communication Architecture—Data Transmit**

***Data Transmit***



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

- the task has permission to access the SCI
- there is no other ongoing send request

Actual data sending does not happen within the scope of SciSend( ). If SciSend( ) returns success for the request, PPSM-GT will handle the SCI interrupts and start sending data in the background. The application will be able to handle other interrupts (for example, pen interrupts) in the foreground.

The calling task 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-GT only transmits data through SCI when the CTS pin is asserted by the receiver.

## Terminating a Send Request

A send request will be terminated under the following circumstances:

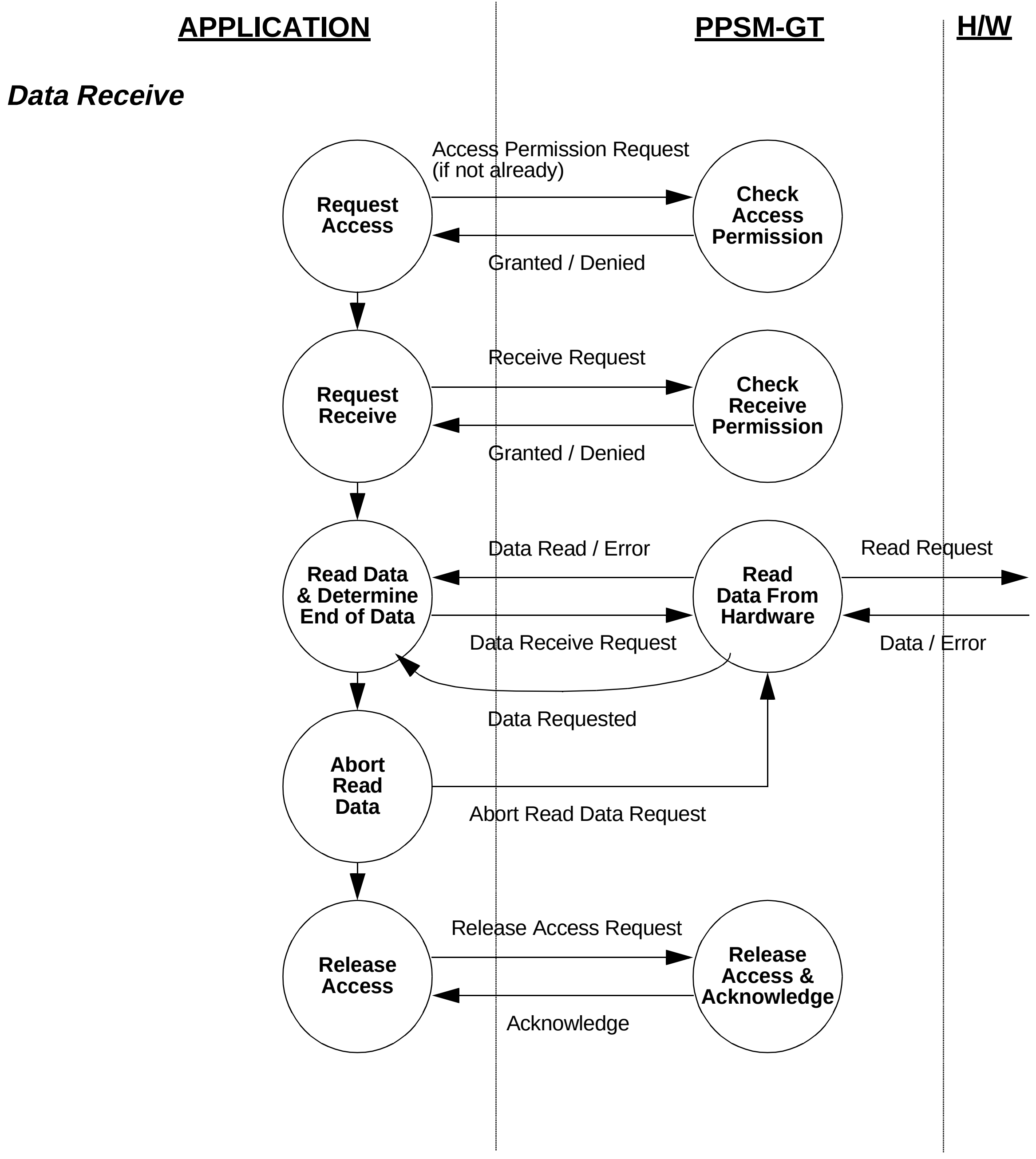
- After PPSM-GT finishes sending all data, it will post an event with the message data SCI_DATA_SENT to the calling task. This marks the completion of the send request.
- If a timed-out error condition occurs during the course of sending data, PPSM-GT will post the event with the message data SCI_ERROR and the corresponding error code. This marks a failed send request, and the calling task should determine the recovery actions. The current transmission is aborted after the time out happened.
- A task aborts the ongoing send request by calling SciSendAbort( ).

The calling task should release the SCI access permission by calling SciUnbindPort( ) or SciClose( ) as soon as it is not needed anymore.

## Initiating a Receive Request

Figure 18.5 shows data reception in the SCI communication architecture.

**Figure 18.5 SCI Communication Architecture—Data Receive**



Tasks can receive data from the SCI by calling SciReceive( ) to initiate receive requests. A receive request will be accepted if both of the following are true:

- the application has permission to access the SCI
- there is no other ongoing receive request

Actual data reception does not happen within the scope of SciReceive( ). If SciReceive( ) returns success for the request, PPSM-GT will handle the SCI interrupts and start waiting for data in the background. The application will be able to handle other interrupts (for example, pen interrupts) in the foreground.

## Reading Received Data

When PPSM-GT has received data from the SCI, it will post an event with the message data SCI_DATA_RECEIVED to the calling task. The calling task should then call SciReadData( ) as soon as possible to read the received data from PPSM-GT.

As PPSM-GT is receiving data from the SCI, the following error conditions may arise:

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

In any of these error conditions, PPSM-GT will post the event with the message data SCI_ERROR and the corresponding error code. These error-related interrupt messages only serve as a notification to the calling application; they do *not* stop PPSM-GT from continuing the receive request. The calling application should determine the appropriate recovery actions.

If RTS, CTS is enabled, the RTS pin is negated when PPSM-GT is running SciReadData( ) and asserted after data reading is complete.

## 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-GT will post an event with the message data SCI_ERROR and the corresponding error code. This marks a failed receive request, and the calling task should determine the recovery actions. The current data reception is aborted after the time out happened.

- A task aborts the ongoing receive request by calling SciReceive( ) with the abort flag.

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

## SCI Resources Interface Constraints

### SCI port

There is a one-to-one relationship between the software and hardware ports. Only one software SCI port can be bound to one hardware UART port at any one time.

### Receive buffer overflow

Application and task swapping is allowed during SCI data transmission and reception. Therefore, in the multitasking environment, it is advisable for the SCI communication task to have a high priority to avoid a receive buffer overflow problem.

## SCI Resource Interface Interrupt Messages

After an application is granted permission to use the SCI, it can initiate a data transmission request. As the data transmission progresses, it will receive the events with the corresponding message data under the circumstances shown in Table 18.2.

When an error condition has occurred, the interrupt message data, SCI_ERROR, appended with an error code will be returned to the calling application. The error codes are shown in Table 18.2.

**Table 18.2 SCI_ERROR Messages Description**

| SCI Message | Description |
|---|---|
| SCI_RX_TMOUT_ERR | For data transmission time-out condition once the transmission has started. |
| SCI_FRAME_ERR | For frame error condition during data receive. |
| SCI_PARITY_ERR | For parity error condition during data receive. |
| SCI_OVERRUN_ERR | For overrun error condition during data receive. |
| SCI_NODATA_ERR | For prematurely requesting PPSM-GT for data before data has been received. |
| SCI_DATA_RECEIVED | Data has been received from the SCI. The interrupt message data will be returned to the calling application. |
| SCI_DATA_SENT | Data send request has been completed. The interrupt message data will be returned to the calling application. |

## Multiple SCI Usage Recommendation

When two or more tasks are required to use the SCI resources, the following recommendations must be observed:

1. Each SCI software port is owned by *only* one task.
2. Each SCI hardware port can be bound by *only* one software port at any one time.
3. Tasks should unbind the software port from the hardware port when communication is complete.
4. Tasks should check for hardware port availability before calling an SCI API to TX or RX.
5. All hardware and software SCI ports must be released to the SCI pool when they not required anymore.

The following series of figures presents a pictorial overview of how the kernel handles two SCI communications. Figure 18.6 through

Figure 18.10 pertain to transmission, and Figure 18.11 through Figure 18.16 pertain to receive.

**Figure 18.6 Task 1 and 2 request the H/W port but only task 1 is successful**

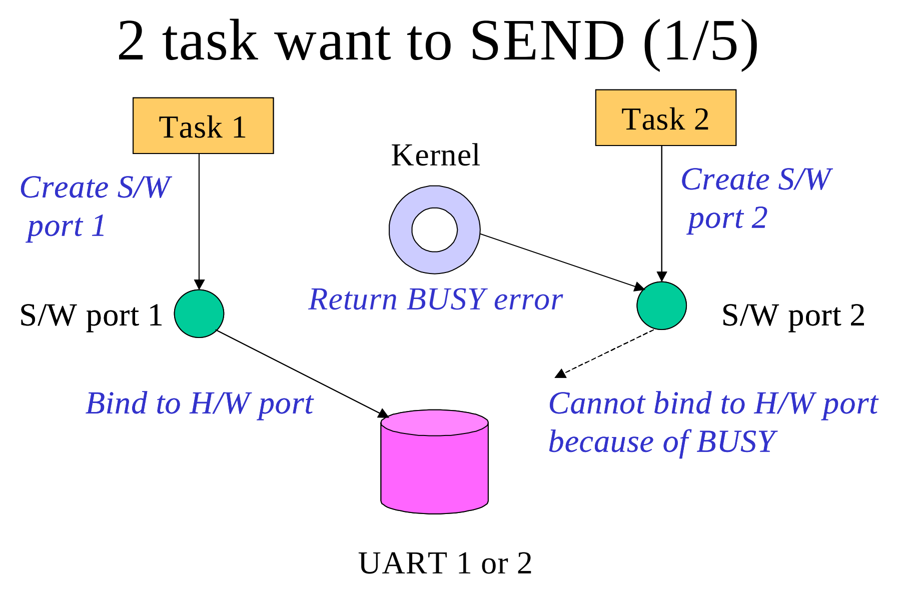


**Figure 18.7 When the H/W port is not available, task 2 polls for the H/W port**

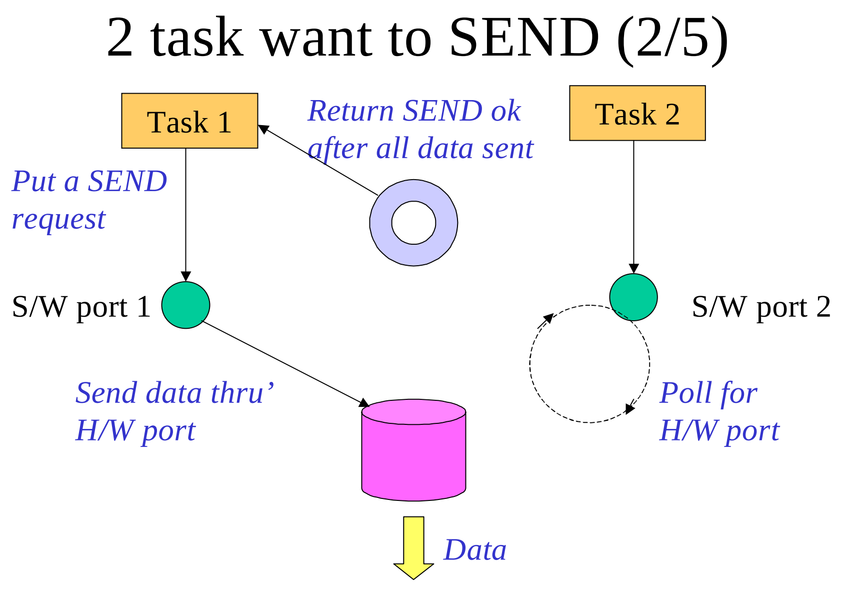


**Figure 18.8 When task 1 has finished transmission, the H/W port is released and task 2 is bound to the H/W port**

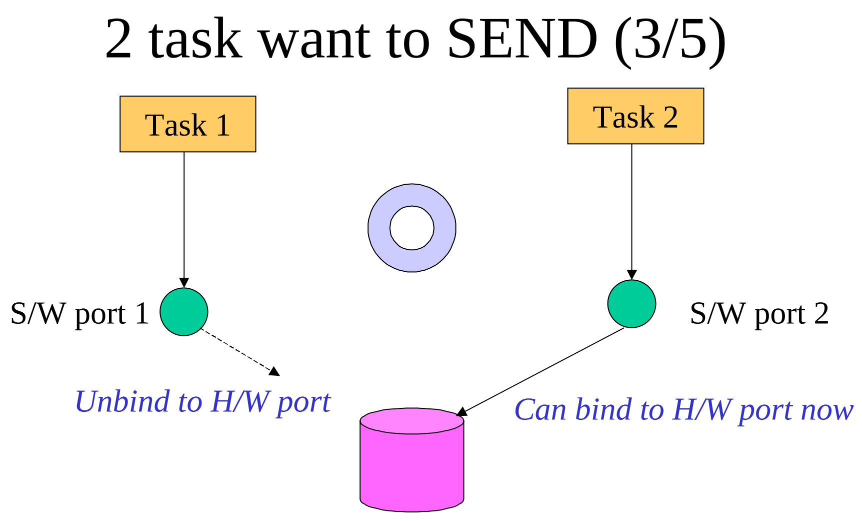


**Figure 18.9 Task 2 sending through the H/W port**

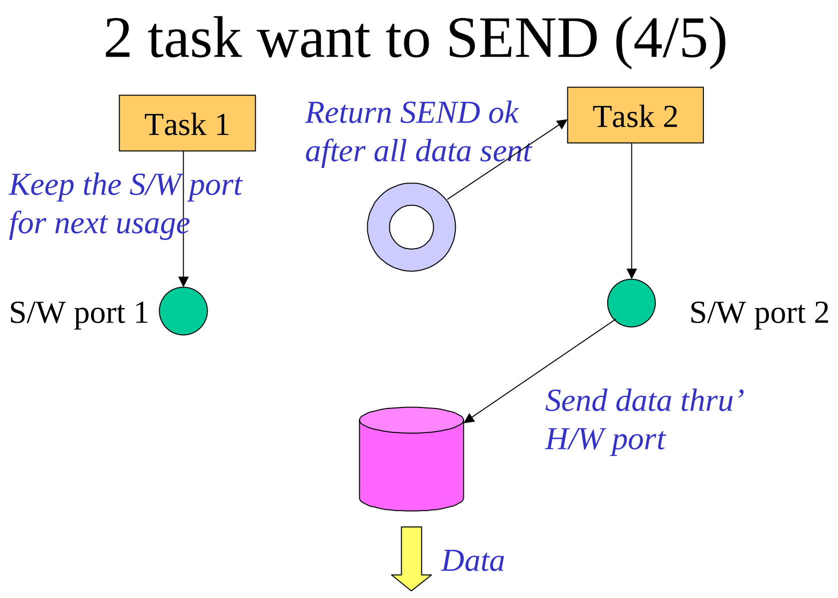


**Figure 18.10 Task 2 also releases the H/W port when transmission is complete**

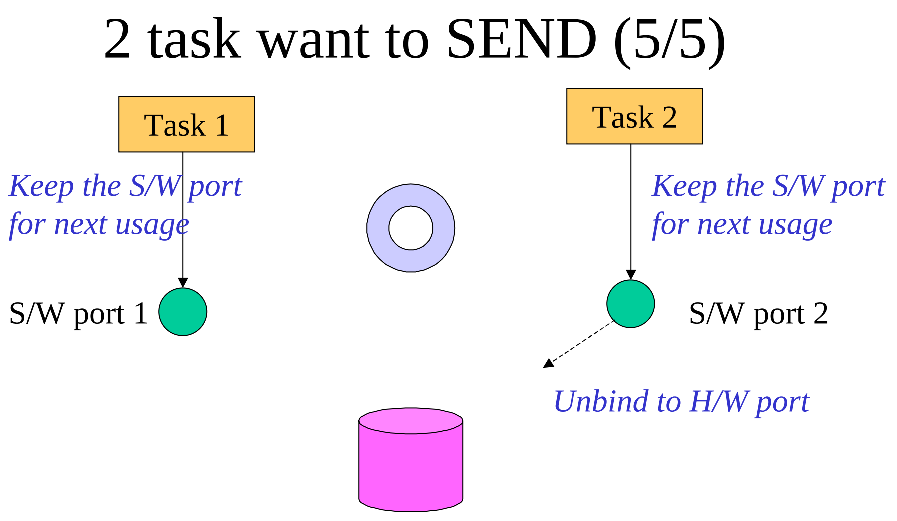

2 task want to SEND (5/5)
Task 1
Task 2
Keep the S/W port for next usage
Keep the S/W port for next usage
S/W port 1
S/W port 2
Unbind to H/W port

**Figure 18.11 Both task 1 and 2 want to receive data from the SCI, but the H/W port has been bound to task 1**

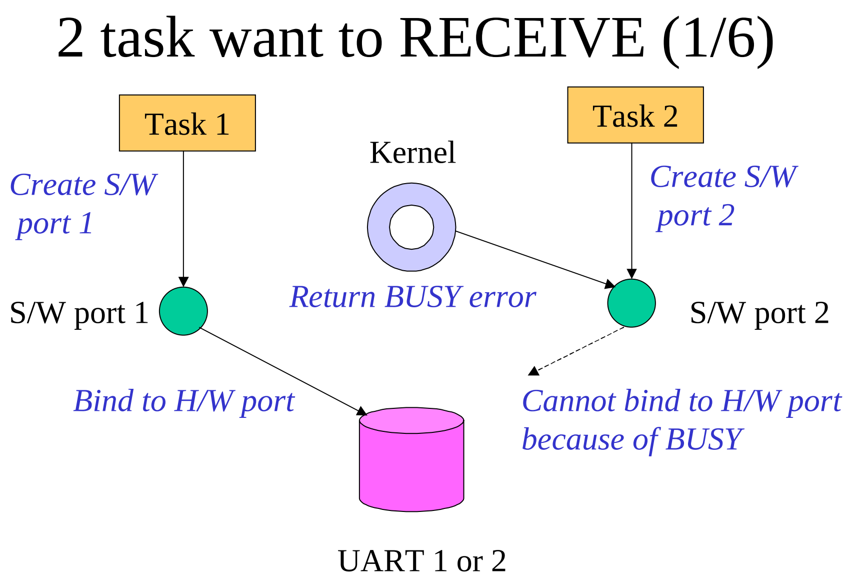


**Figure 18.12 Task 1 gets to receive while task 2 waits for the H/W port**

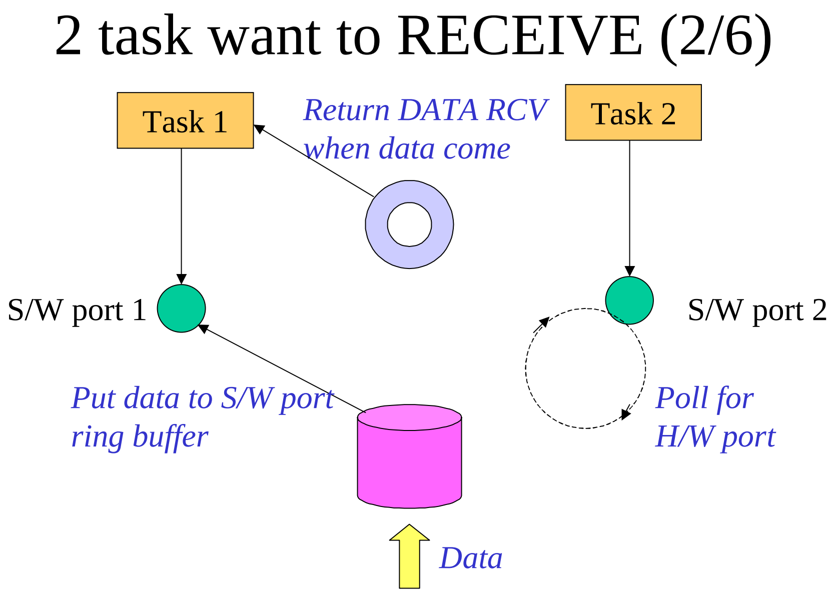

2 task want to RECEIVE (2/6)
Task 1
Return DATA RCV when data come
Task 2
S/W port 1
S/W port 2
Put data to S/W port ring buffer
Poll for H/W port
Data

**Figure 18.13 Task 1 calls SciReadData to read data from buffer**

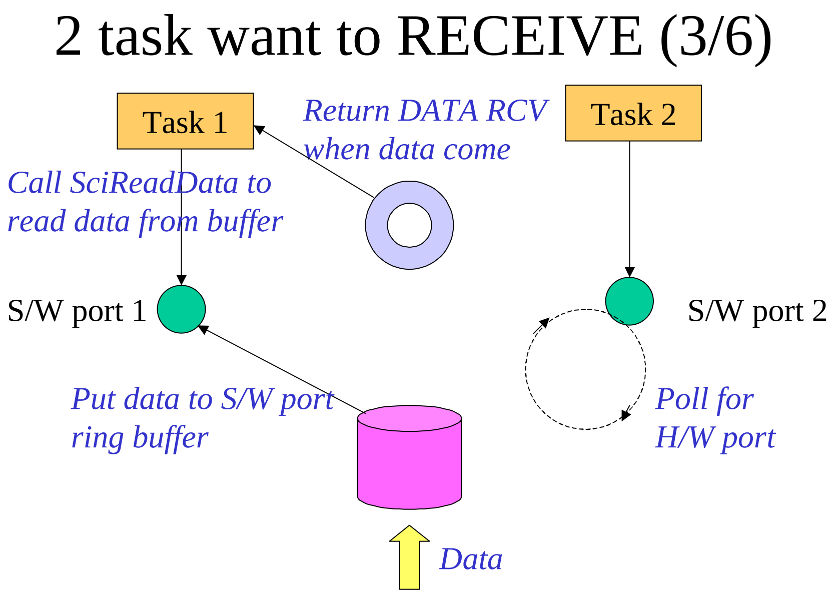

2 task want to RECEIVE (3/6)
Task 1
Return DATA RCV when data come
Task 2
Call SciReadData to read data from buffer
S/W port 1
S/W port 2
Put data to S/W port ring buffer
Poll for H/W port
Data

**Figure 18.14 Task 1 releases the H/W port when reception is complete**

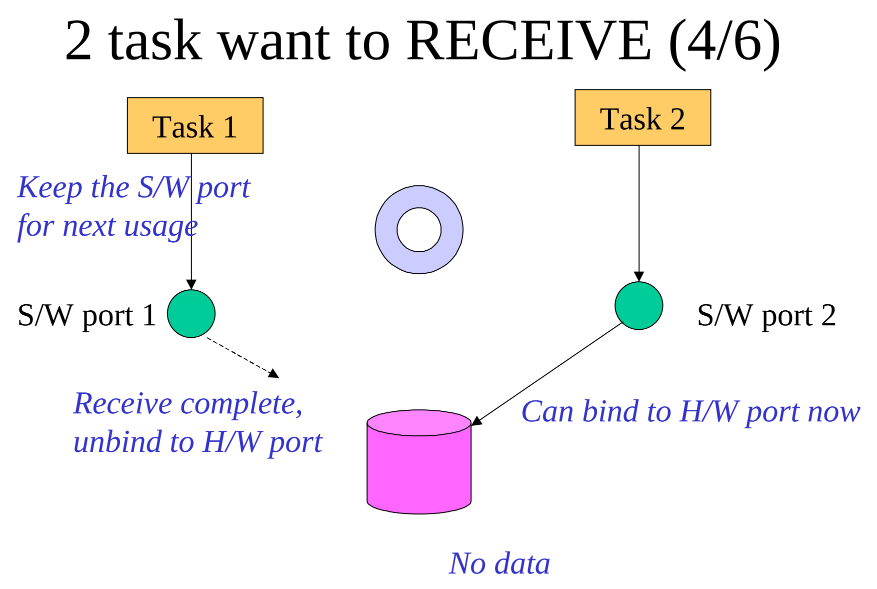

2 task want to RECEIVE (4/6)
Task 1
Task 2
Keep the S/W port for next usage
S/W port 1
S/W port 2
Receive complete, unbind to H/W port
Can bind to H/W port now
No data

**Figure 18.15 Task 2 gets to bind with the H/W port to receive data**

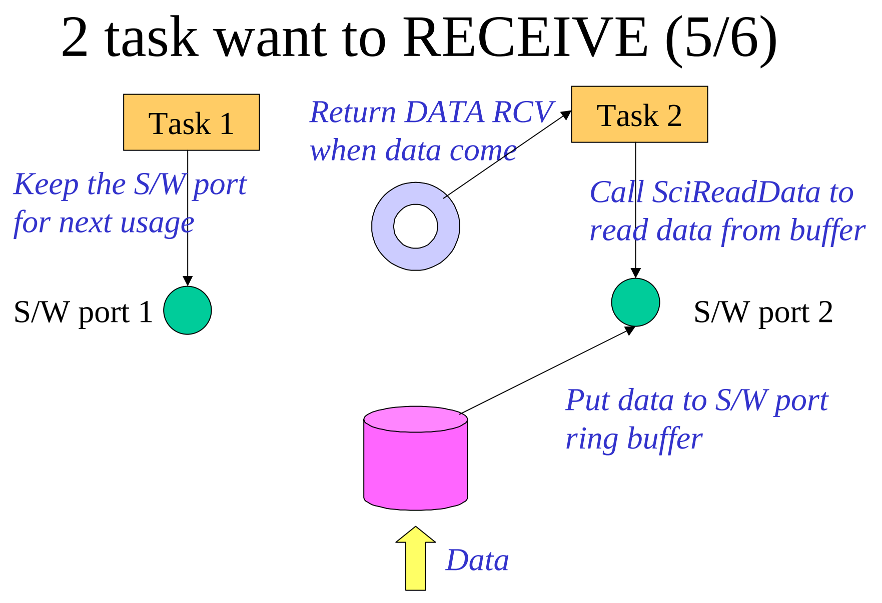
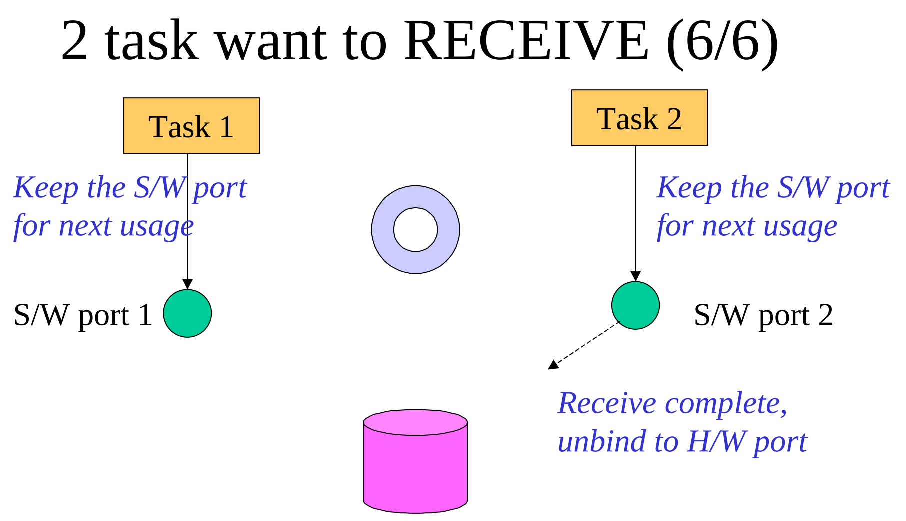

2 task want to RECEIVE (5/6)
Task 1
Return DATA RCV when data come
Task 2
Keep the S/W port for next usage
Call SciReadData to read data from buffer
S/W port 1
S/W port 2
Put data to S/W port ring buffer
Data

**Figure 18.16 Task 2 also releases the H/W port when reception is complete**


2 task want to RECEIVE (6/6)
Task 1
Task 2
Keep the S/W port for next usage
Keep the S/W port for next usage
S/W port 1
S/W port 2
Receive complete, unbind to H/W port

## Programming Using SCI Services

### Requesting for SCI

| | |
|---|---|
| STATUS **SciCreate**(P_SCI_PORT_ID pPortId) | Based on the availability, the system will respond. SciCreate( ) will create a software SCI port. |
| STATUS **SciBindPort**(SCI_PORT_ID portId, U8 uartPort) | If an SCI resource is available, SciBindPort( ) will register the software SCI port to the hardware UART port and the calling task. If no SCI resource is available, an error message of ERR_SCI_PORT_BUSY is returned. |
| STATUS **SciOpen**( SCI_PORT_ID portId) | SciOpen( ) will create and register the software SCI port at the same time. It performs the functions of SciCreate() and SciBindPort(). |

## Configuring the SCI

| | |
|---|---|
| STATUS **SciConfig**(SCI_PORT_ID portId, U8 mode, U8 baudRate, U8 parity, U8 stopBits, U8 charLen) | Tasks can use SciConfig( ) to reconfigure the SCI to the required settings. Any ongoing 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 18.3* shows the list of configurations and settings supported along with the corresponding selection flag to be used with SciConfig( ). (Refer to the appropriate hardware manual for details.)

**Table 18.3 SCI Configuration Table**

| Configuration | Supported Settings |
|---|---|
| Operating Mode | Normal NRZ mode<br>IrDA mode |
| Baud Rate | 600 bps<br>1200 bps<br>2400 bps<br>4800 bps<br>9600 bps<br>14400 bps<br>19200 bps<br>28800 bps<br>38400 bps<br>57600 bps<br>115200 bps |
| Parity | No parity<br>Odd parity<br>Even parity |
| Number of Stop Bits | 1 stop bit<br>2 stop bit |
| Character Length | 7-bit character<br>8-bit character |

## Inquiring about the SCI Configurations

| | |
|---|---|
| STATUS **SciGetConfig**(SCI_PORT_ID portId, P_U8 pMode, P_U32 pBaudRate, P_U8 pParity, P_U8 pStopBits, P_U8 pCharLen) | SciGetConfig( ) provides the interface for applications to inquire about the current configuration settings of the SCI. It returns the selection flag for the corresponding configuration setting as shown in *Table 18.3*, except for baud rate. The actual baud rate in bps is returned instead. |

## Inquiring about the SCI CTS and RTS Status

| | |
|---|---|
| STATUS **SciCtsStatus**(SCI_PORT_ID portId) | Returns the CTS status with regard to the API called. Note that the status may not be up to date due to time delays between function calls. |
| STATUS **SciRtsStatus**(SCI_PORT_ID portId) | Returns the RTS status with regard to the API called. Note that the status may not be up to date due to time delays between function calls. |

## Setting Data Transmission and Reception Time Out

| | |
|---|---|
| void **SciSetTimeout**(P_SCI_TMOUT pTmout, TICK timeout, EVTTYPE portId) | The data transmission time out is defined to be the time interval between two hardware SCI interrupts. This time out is set to safeguard the application from deadlocking itself when the data stream terminates unexpectedly. |

If RTS, CTS is enabled, after calling SciSend( ) to initiate transmission, an application will receive a time-out error if CTS is not asserted within the time-out period. If CTS is asserted, an application will receive a time-out error if the time interval between two hardware SCI interrupts is larger than the time-out period.

The range of time-out values supported is 0 to 60,000.

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

## Setting Data Transmission Delay

| | |
|---|---|
| STATUS **SciSetDelay**(SCI_PORT_ID portId, U8 type, U16 delay) | In order to communicate with an application in PC, such as HyperTerminal and Telex, transmitting data in a burst of pulses periodically would greatly increase the accuracy of transmission. This function allows a user to set a delay, in units of 1 ms, between each transmission of all data in transmit FIFO (between two hardware interrupts). |

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

- SCI_TXDELAY_CLEAR means clear the delay during transmission.
- 1 to 60,000 means allowing the delay interval between two hardware SCI interrupts to be 1 millisecond to 1 minute.

## Sending Data to the SCI

| | |
|---|---|
| STATUS **SciSend**(SCI_PORT_ID portId, U8 sendFlag, P_U8 pSendData, U32 dataLen) | SciSend will send data that is stored in the pSendData buffer through an SCI port when called. After the data transmission is complete, the system will inform the calling task through an event. |

## Controlling Sending of Data

| | |
|---|---|
| STATUS **SciSendCtrl**(SCI_PORT_ID portId, U8 controlType) | Tasks can pause data transmission of the SCI when hardware flow control is enabled. |

SciSendCtrl(portId, SCI_RCTS_PAUSE) will cause a data transmission to be stopped.

The error code ERR_SCI_NO_CTRL is returned if hardware flow control is not enabled.

Tasks can resume a data transmission by calling SciSendCtrl(portId, SCI_RCTS_CONT).

This API should be used instead of SciSendAbort( ) to pause data transmission.

## Terminating a transmission

| | |
|---|---|
| STATUS **SciSendAbort**(SCI_PORT_ID portId, U8 abortFlag, P_U8 *ppSendData, P_U32 pSendSize) | This routine aborts the current SCI transmission. An abort is not a pause. The aborted transmission cannot be continued via SciSendCtrl( ). To resend, call SciSend( ) again. |

## Setting the FIFO level

| | |
|---|---|
| STATUS **SciSetFifoLevel**(SCI_PORT_ID portId, U8 fifoFlag, U8 fifolevel) | This routine is used to set the UART FIFO level mark. |

## Receiving Data from the SCI

| | |
|---|---|
| STATUS **SciReceive**(SCI_PORT_ID portId, U8 receiveFlag) | SciReceive( ) will enable data reception from the SCI port. PPSM-GT will receive the data from the SCI port and transfer it into the receive buffer of the SCI port. When there is data in the receive buffer, the system will inform the calling task by an event. The system does not send out the event continuously. When data is received, the system will inform the task once. When the task completes a SciReadData( ) operation but there is still un-read data inside the receive buffer, the system will send out another event to notify the task. This process continues until all data in the receive buffer is read out to the calling task. |
| STATUS **SciReadData**(SCI_PORT_ID portId, P_U8 pData, U16 bufSize, P_U16 pSizeRead) | The calling task should use SciReadData( ) to transfer the data from the receive buffer to the local buffer as soon as possible to avoid receive buffer overflow. |

## Changing the Receive Buffer Size

| | |
|---|---|
| STATUS **SciSetRxBufSize**( SCI_PORT_ID portId, U16 newSz) | This routine allows tasks to change the receive buffer size to a user-defined size. The default size is 256 bytes. |

## Controlling Receiving of Data

| | |
|---|---|
| STATUS **SciRcvCtrl**(SCI_PORT_ID portId, U8 controlType) | After RTS, CTS hardware flow control is enabled, PPSM-GT automatically pauses data reception once the internal SCI buffer (not FIFO) is full. Data reception is resumed after data is read out by SciReadData( ) in the application. If the interval of CTS remaining negated is longer than the time-out period, a time-out error will occur. |

Tasks can pause data reception of the SCI when hardware flow control is enabled. SciRcvCtrl(portId, SCI_RCTS_PAUSE) allows data reception to be paused at SCI port 1.

The error code ERR_SCI_NO_CTRL is returned if hardware flow control is not enabled.

Tasks can resume data reception by calling SciRcvCtrl(portId, SCI_RCTS_CONT).

## Controlling SCI hardware flow control

| | |
|---|---|
| STATUS **SciFlowCtrl**(SCI_PORT_ID portId, U8 controlType) | Set the hardware flow control. |

- Enabling RTS, CTS hardware flow control:

  Using this API, a task could enable RTS, CTS hardware flow control by setting the controlType to SCI_RCTS_ENABLE.

  If hardware flow control is not enabled when calling RTS, CTS flow control APIs, the error code ERR_SCI_RCTS_IDLE is returned to the application.

  RTS is asserted after RTS, CTS flow control is enabled.

- Disabling RTS, CTS hardware flow control:

  Disable RTS, CTS hardware flow control by setting the controlType to SCI_RCTS_DISABLE.

  The system negates the RTS pin immediately after hardware flow control is disabled. Any further changes in RTS or CTS pins are ignored by the system.

  RTS is negated after RTS, CTS flow control is disabled.

## Clearing the SCI Transmit and Receive Buffer

| | |
|---|---|
| STATUS **SciFlushFifo**(SCI_PORT_ID portId, U8 fifoFlag) | This routine will flush the hardware transmit or receive FIFO buffer depending on the input for fifoFlag. The flag is set for RX for receive buffer and TX for transmit buffer. All data on the TX or RX FIFO buffer will be destroyed. |

**NOTE** The buffer referred to here is the hardware RX FIFO buffer or hardware TX FIFO buffer, not the software receive buffer or software transmit buffer of the SCI port.

## Clearing the SCI Receive Buffer

| | |
|---|---|
| STATUS **SciFlushRxFifo**(SCI_PORT_ID portId) | This routine will flush the RX FIFO buffer, and all data on the RX FIFO buffer will be destroyed. |

## Terminate the SCI port

| | |
|---|---|
| STATUS **SciUnbindPort**(SCI_PORT_ID portId) | Unbind SCI hardware port from the task |
| STATUS **SciClose**(SCI_PORT_ID portId) | Close the SCI port |

These routines will cause the system to unbind the software port from the hardware port. The SCI port is then free to be used by other tasks. The difference between SciUnbindPort( ) and SciClose( ) is that SciUnbindPort( ) will leave the software SCI port with the same configuration that can be used later, while SciClose( ) will delete the software port and free all memory allocated to it.

Tasks should not hold onto the hardware port if they do not need it.

## Summary

The PPSM-GT SCI services provide two methods for accessing the SCI resources: the Three Steps and Five Steps Methods.

The Three Steps Method enables applications to access the SCI resources through straightforward and easy-to-use APIs. This method is suitable for an application that does not need to save the SCI configuration for future use.

The Five Steps Method is an expanded version of the Three Steps Method. It breaks down the three steps to five steps to allow an application to save the SCI configuration and hold onto the SCI ports for future usage.

## Code Examples

**Listing 18.1 Programming SCI services using the three steps method**

```
/* Variable Definition*/

SCI_PORT_IDportId;
.....

/* Initiating a Send request*/
  SciOpen(&portId, UART_2);
  SciConfig(portId, SCI_SERIAL_MODE, SCI_38400_BPS, NO_PARITY,
ONE_STOP_BIT, EIGHT_BIT_CHAR);
  SciSend(portId, SCI_SEND_REQUEST, &SendMsg, gSendDataLen);
.....
```

```
/* Terminating a transmission*/
.....
  /* Abort send. Store the pointer of send buffer and no. of bytes
have been..*/
  /* .. sent in gpSendbuf and gSendbyte respectively */
    SciSendAbort(portId, SCI_SEND_ABORT, &gpSendbuf, &gSendbyte);

    SciClose(portId);
  .....
```

**Listing 18.2 Programming SCI services using the five steps method**

```
/* Variable Definition*/

SCI_PORT_IDportId;

.....

/* Initiating a Send request*/
  SciCreate(&portId);

  status = SciBindPort(portId, UART_2);

  SciConfig(portId, SCI_SERIAL_MODE, SCI_38400_BPS, NO_PARITY,
ONE_STOP_BIT, EIGHT_BIT_CHAR);
  SciSend(portId, SCI_SEND_REQUEST, &SendMsg, gSendDataLen);
.....

/* Terminating a transmission*/
.....
  /* Abort send. Store the pointer of send buffer and no. of bytes
have been..*/
  /* .. sent in gpSendbuf and gSendbyte respectively */
    SciSendAbort(portId, SCI_SEND_ABORT, &gpSendbuf, &gSendbyte);
    status = SciUnbindPort(portId);
.....

    SciClose(portId);
  .....
```

# 19

# IrDA Management Services

As infrared data communications, based on standards from the Infrared Data Association (IrDA), become widely available on embedded devices, there is a growing need for embedded systems to support IrDA services as a standard feature.

The IrDA management services in PPSM-GT are a layered set of protocols particularly aimed at point-to-point infrared communications and the applications needed in that environment.

This chapter is organized into the following main sections:

- IrDA Management Services Fundamentals
- Programming Using IrDA Management Services
- Summary
- Code Examples

# IrDA Management Services Fundamentals

**Figure 19.1 IrDA Management Services Architecture**

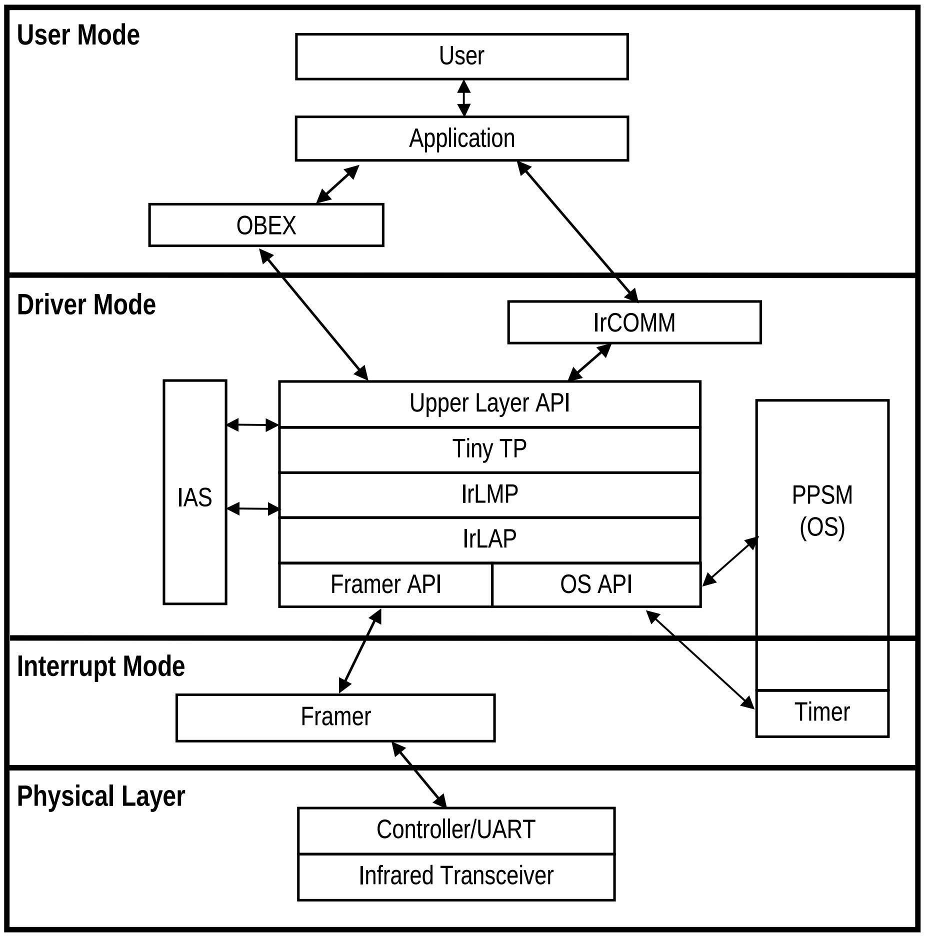


Figure 19.1 shows the IrDA management services architecture, which can be divided into the following layers:

- Physical Layer
- Interrupt Mode Layer
- Driver Mode and User Mode Layers

## Physical Layer

The physical layer includes the optical transceiver. This layer deals with shaping and other characteristics of infrared signals, including the encoding of data bits, and some framing data such as begin of frame and end of frame flags (BOFs and EOFs) and cyclic redundancy checks (CRCs). This layer must be at least partially implemented in hardware, and in some cases it is handled entirely by hardware.

Some level of customization may be required in the two functions IrdInitTransceiver() and IrdShutDownTransceiver() when supporting different IrDA transceivers. Information—such as how to shut it down, power it up, and set various power modes—needs to be provided. The function names must be named exactly as specified in Chapter 6, Table 6.1.

## Interrupt Mode Layer

The interrupt mode layer is responsible for handling UART interrupts and building up a received frame as well as for handling the transmission of an IrDA frame down to the hardware.

It also acts to isolate the remainder of the stack from the ever-changing hardware layer. Its primary responsibility is to accept incoming frames from the hardware and present them to the driver mode layer.

## Driver Mode and User Mode Layers

The driver mode and user mode layers form the core of the IrDA management services. IrLAP, IrLMP, IAS, TinyTP, and IrCOMM protocols are grouped under driver mode layer, whereas IrOBEX is under the user mode layer protocol. A brief description of each protocol is in Table 19.1.

**Table 19.1 IrDA Protocols**

| IrDA Protocol | Description |
|---|---|
| IrLAP | Link Access Protocol. Establishes the basic reliable connection. |
| IrLMP | Link Management Protocol. Multiplexes services and applications on the LAP connection. |
| IAS | Information Access Service. Provides a "yellow pages" of services on a device. |
| TinyTP | Tiny Transport Protocol. Adds per-channel flow control to keep things moving smoothly. This is a very important function. |
| IrOBEX | The Object Exchange protocol. Easy transfer of files and other data objects. |
| IrCOMM | Serial and Parallel Port emulation, enabling existing applications that use serial and parallel communications to use IR without change. |

## IrDA Parameters

During the initiation of a connection, only those parameters shown in Table 19.2 can be used in setting up the connection. These parameters represent the range of IrDA 1.1 negotiable parameters in PPSM-GT.

**Table 19.2 IrDA initialization parameters**

| Parameter | Allowed Values |
|---|---|
| Baud Rate | 9600 bps, 19200 bps, 38400 bps, 57600 bps and 115200 bps |
| Data Size | 64, 128, 256, 512, 1024, and 2048 bytes |
| Window Size | 1 |
| Additional BOFs | 0, 1, 2, 3, 5, 12, 24, and 48 |
| Maximum Turnaround Time | 500, 250, 100, and 50 ms |
| Disconnect/Threshold Time | 40 seconds |

## IrDA Services

There are two IrDA services provided by PPSM-GT:

- IrCOMM Services
- IrOBEX services

### IrCOMM Services

IrCOMM services are IrDA services that were designed to provide serial emulation to legacy applications. These services enable the applications to communicate with a peer device over an IrDA infrared link instead of the wired link.

Typically, an IrCOMM implementation consists of two parts:

- The IrCOMM protocol, which is independent of any communications API, handles the location of, connection to, and communications with other IrCOMM devices. It only takes care of the way that information is exchanged over the infrared link.
- Some sort of port emulation entity that provides a way for the native communication API of the target operating system to use the IrCOMM protocol.

### *Port Emulation*

There are four service types defined in the IrCOMM specification for port emulation. They are:

1. 9-Wire (serial emulation only): uses control channel for status of standard RS-232 non-data circuits. Uses TinyTP.
2. 3-Wire (parallel and serial emulation): Minimal use of control channel. Uses TinyTP.
3. 3-Wire Raw (parallel and serial emulation): Send data only; no non-data circuit information and hence no control channel. Run directly on IrLMP.
4. Centronics (parallel emulation only): Uses control channel for status of Centronics non-data circuits. Uses TinyTP.

The 9-Wire, 3-Wire and Centronics are known as “cooked” service types in the IrDA IrCOMM specification. The term “cooked” refers to the fact that the service runs on TinyTP and provides a mechanism for sending setup and status information over a “control” channel.

The 3-Wire raw service is “raw” because it provides no control channel and runs directly on top of IrLMP.

PPSM-GT IrDA services supports only the emulation of the 9-Wire service because the IrCOMM 9-Wire also provides all of the functionality of the IrCOMM 3-Wire service and the IrCOMM 3-Wire Raw. The Centronics services are not supported.

### *IrCOMM 9-Wire*

IrCOMM 9-wire is the most common among the four services. It provides the ability to send and receive the status of the non-data circuits of the RS-232 interface (DTR, RTS, CTS, DSR, RI, and CD). These circuits are normally used by the Data Terminal Equipment (DTE) to control the Data Circuit-Terminating Equipment (DCE). A DTE is usually a computer, and a DCE is usually a modem.

This implementation of the 9-Wire service can be compiled as a DCE or DTE. As a DTE, communication with both a DCE and another DTE is possible. Null modem emulation is used to translate DTE circuits to DCE circuits. As a DCE, only communication with a DTE is possible. Any DCE line settings that are received are ignored. In addition, status information on the non-data circuits is provided.

### IrOBEX services

The IrOBEX services are an implementation of the IrDA Object Exchange specification (IrOBEX) for IrDA protocol stacks. They provide the ability to Put data objects very simply and flexibly, thereby enabling rapid application development and interaction with a broad class of devices including PCs, PDAs, data collectors, and cameras.

IrOBEX makes it extremely simple for one or more applications within a device to send or receive data objects. The data objects can be files, images, records, or any unit of data that makes sense to the applications involved. Applications can avoid the usual complex programming tasks associated with IR—performing discovery, making connections and managing transactions. Instead they make a simple call to send (as if dropping the object in the mail), and they are notified when a complete incoming object is ready to process (as if it had arrived in the mail).

IrOBEX runs on top of the IrDA protocol stack. The rest of this chapter gives a brief introduction to the IrOBEX service.

#### *Execution Flow*

The IrOBEX service is designed to be both event driven and user interface driven, and it will process each until completion. Events are generated by the IrOBEX protocol for a variety of reasons, such as when connections are established, when IrOBEX commands are received or when API requests are complete. These events are usually originated by some event coming from the IrDA protocol stack.

The User Interface initiates IrOBEX Client requests through the IrOBEX Layer API. The IrOBEX Layer is not re-entrant, which means that when it is processing an event, it cannot be called to process another event or deal with a call from the application. There are some exceptions where it is permissible to call down into the protocol while in the context of an application callback. These situations are identified in the function references.

The flow of execution is different for the Client and Server Applications because of the initiating event. Server applications are usually idle, waiting for a connection to be established or a request received from a client. The client is initiated by an IrOBEX Layer API call, which is usually originated by a user interface request.

The following list identifies the likely flow of execution for a client operation.

1. The user requests that an object be Sent (Put) via the application's User Interface.
2. The application calls the IrOBEX Layer API, requesting a connection to a peer device. The protocol (specifically the Stack Interface Layer) performs device discovery and IAS lookups and establishes a TinyTP Connection to a peer device. Once this process is complete, the IrOBEX Layer notifies the application via the application's event handler (callback function) with the status of the request (pass / fail).
3. The application then often requests that the IrOBEX Layer API create an IrOBEX protocol connection to the peer IrOBEX application. When the connection request is complete, the application's event handler is again notified with the result of its request.
4. The application can then begin exchanging objects by calling the IrOBEX Layer API to request that an IrOBEX Operation be performed (Put). The IrOBEX operation will take place.
5. Throughout the exchange, the Object Store is instructed to either retrieve or store the Object data being transferred by the Command Interpreter. Each time a sent IrPacket is returned by the IrDA stack, the Stack Interface Layer queries the Command Interpreter via the Packet Parser to see if it has more data to send (which it reads from the Object Store). This behavior continues until there is no more data to send (the object is completely transferred).

Once the object exchange is complete, the application's event handler is notified with the result of the exchange.

The application can then call the IrOBEX Layer API to exchange more objects. Or, if it is complete, it can instruct IrOBEX to disconnect from the peer device.

The likely flow of operation for a Server is as follows:

1. An incoming TinyTP connection to the Server is accepted by the Stack Interface from the IrDA stack. The Stack Interface Layer then notifies the application of the incoming connection via the application’s event callback.
2. IrOBEX packets begin to arrive at the Stack Interface Layer and are passed to the Packet Parser. It validates them and passes important information on to the Command Interpreter. The Command Interpreter then indicates the operation to the application via its event callback. The application can then either accept or reject the operation by calling the IrOBEX Layer API. The application can also call the IrOBEX Layer API to retrieve information, such as header contents, to help it determine if the operation should be accepted or rejected.
3. Once accepted the Command Interpreter calls the Object Store write and read functions to exchange the object’s data as it arrives or is needed for transmit. It then calls the Packet Parser, which passes protocol data to the Stack Interface layer for inclusion in packets, which are sent via the IrDA stack to the peer device. The server application is not responsible for managing the exchange of object data. Once it has accepted the IrOBEX request, the protocol takes over the exchange until it is complete.
4. Object exchanges that take multiple TinyTP packets to perform use the same method as described previously for the Client. When a sent IrPacket is returned by the IrDA stack, the Stack Interface Layer calls through the Packet Parser to the Command Interpreter to retrieve more data. It, in turn, calls the Object Store to read the data. This behavior continues until the object is completely transferred.
5. Once complete, the Command Interpreter notifies the application's event handler with the status of the exchange. At this time the server usually returns to an idle state, waiting for more IrOBEX requests or for the client to disconnect.

## Programming Using IrDA Management Services

There are also four sets of services available when programming using IrDA management services. They are:

- IrDA Physical Layer Services
- IrCOMM Layer Services
- OBEX Application Layer APIs

### IrDA Physical Layer Services

IrDA physical layer services are mainly used when initializing the system. The APIs are provided to configure and initialize the IrDA hardware. Table 19.3 shows the physical layer services.

**Table 19.3 Physical Layer APIs**

| Physical Layer API | Description |
|---|---|
| STATUS **IrdSetDeviceInfo**(P_U8 info, U8 len) | Set the XID info string used during discovery to the given string and length. The XID info string contains hints and the nickname of the device. The size cannot exceed IR_MAX_DEVICE_INFO bytes. Customized by the manufacturer, depending on the product name. Usually executed during initialization. |
| STATUS **IrdInit**(U8 port) | Starts up the PPSM-GT IrDA_Process task, initializes the appropriate IrDA variables, and initializes the framer functions. The port variable would be used to indicate whether to set up the ISR for UART1 or UART2. This function would ordinarily be set up by the manufacturer during initialization, or when the user enables IR mode from the user interface. Returns a value indicative of the result of the action. |

| Physical Layer API | Description |
|---|---|
| STATUS **IrdDeInit**(void) | This disables the IrDA_Process task and releases variables. This function would also deactivate and release the UART_Isr. |
| STATUS **IrdSetMaxTurnAroundTime**(U8 MaxTat) | Called to set the maximum turnaround time during setup. This is used during the negotiation phase. This function should only be called after the IRDA Process / global variables are initialized. |

## IrCOMM Layer Services

The IrCOMM Layer APIs provide a simple Open, Read, Write, and Close API for sending and receiving data. Typically, the application interface to IrCOMM would be through PPSM-GT system API calls, such as IrcOpen() and IrcRead(). The IrCOMM service also provides APIs to send and receive the status of the non-data circuits of the RS-232 interface (DTR, RTS, CTS, DSR, RI, and CD). Table 19.4 shows the IrCOMM APIs.

**Table 19.4 IrCOMM APIs**

| IrCOMM API | Description |
|---|---|
| void **IrcSetFormat**(U32 format) | Set the format bit-field. This bit-field contains IrCOMM control parameters such as baud rate, parity, character size, and state information on the leads. These are all encoded into a 32-bit bit-field. Format information is sent immediately if data issued by IRCOMM_Write() is pending. If IRCOMM_Write() is called immediately after setting the format, then the format information is sent with the data. If data is pending when IRCOMM_SetFormat() is called, then the format information will be sent after all data has been sent. When format information is sent, the application will be notified with the IRCOMME_STATUS_SENT event. If a field in the format bit-field is changed multiple times before a IRCOMME_STATUS_SENT event, only the last change will be sent. |

| | |
|---|---|
| void **IrcGetFormat**(P_U32 format) | Get the format bit-field. This bit-field contains IrCOMM control parameters such as baud rate, parity, character size, and state information on the leads. |
| void **IrcGetStatusEventCause**(P_U16 eventCause) | Returns into the eventCause variable the reason that an IRCOMME_STATUS_CHANGE event was received by the notify function. Whenever this event is received by the notify function, the cause of the event should be checked by a call to this function. |
| BOOL **IrcIsDeviceBusy**(void) | Returns TRUE if the other device is busy. In the case of a printer, if the busy condition lasts for more than 20 seconds, it may be out of paper. |

### *Opening the Port*

| | |
|---|---|
| STATUS **IrcOpen**(TASK_ID AppCallback) | Open the IrCOMM driver. |

The IrCOMM driver is opened by calling the function IrcOpen(). This function takes one parameter, which is a TASK_ID of the application.

Events are sent by the IrCOMM driver to notify the application of certain events such as:

- when data is available,
- when the driver is ready to write more data,
- when the driver is closed,
- when status information has been received, and
- when status information has been sent.

IrcOpen() initializes the IrCOMM driver and then attempts to discover a device. If there is a device supporting an IrCOMM service, then an attempt is made to connect to the highest level service.

When the connection is completed, the user application is notified with an EVT_IRD_IRCOMME_WRITE event. At that time, the driver is ready to accept data.

Only one discovery is attempted when IrcOpen() is called. If no device is discovered (no EVT_IRD_IRCOMME_WRITE event was received), then it is possible to retry the discovery by calling IrcWrite(). When the connection is down, IrcWrite() will return ERR_IRC_STATUS_FAILED, but it will also attempt to re-establish the connection. When the connection is finally made, the notify function will receive an EVT_IRD_IRCOMME_WRITE event.

### Writing Data

| | |
|---|---|
| STATUS **IrcWrite**(P_U8 buff, U16 len) | Write a buffer of data to the IrCOMM driver. A buffer can be any size. If the buffer size is greater than the maximum transmit size, then the IrCOMM driver will break the data up and send the data out in units equal to the maximum transmit size until the entire buffer has been sent. The data is not actually sent when the function is called, but is managed by the IrCOMM driver and will be sent at the appropriate time. If a connection does not exist, then the IrCOMM driver will establish one. |

Data is written to IrCOMM using the IrcWrite() function. This function accepts two parameters. The first is a pointer to the block of data that is to be written, and the second is the size of the data. The block of data is allocated by the application and can be of any size.

The IrCOMM driver will break the data up into packets of the size negotiated by IrLAP and send them until the entire buffer has been sent. If IrcWrite() returns ERR_IRC_STATUS_PENDING, then the buffer cannot be used until the IrCOMM driver has completed sending.

When the buffer is sent, the driver will notify the application with an EVT_IRD_IRCOMME_WRITE event. At that time, the buffer is free and the driver can accept more data. If IrcWrite() returns ERR_IRC_STATUS_SUCCESS, then the buffer is free to be used immediately. If IrcWrite() returns ERR_IRC_STATUS_FAILED, then driver could not accept the data because it was not in the correct state.

## Reading Data

| | |
|---|---|
| U16 **IrcRead**(P_U8 buff, U16 len) | Read data from the IrCOMM driver into a buffer. The actual amount of data read is returned. If no data exists, then 0 is returned. If the buffer is not big enough to receive all the available data, the amount of data actually read is returned and subsequent calls to IRCOMM_Read() will retrieve more data. When the data has all been read, then 0 will be returned on subsequent calls. |

Data is read from IrCOMM using the IrcRead() function.

When the application is notified of an EVT_IRD_IRCOMME_READ event, data is available to be read. IrcRead() requires two parameters. The first is a buffer into which data will be copied, and the second is the size of the buffer.

When IrcRead() is called, the driver copies the available data into the buffer provided by the application. IrcRead() will return the amount of data that was actually copied. If the buffer is not big enough to receive the data held by the driver, then the actual amount copied will be returned and the application can call IrcRead() again to retrieve more data.

## Closing the Port

| | |
|---|---|
| STATUS **IrcClose**(void) | Close the IrCOMM driver. If an IrDA connection exists, it will be disconnected. If write data is pending, it will be lost. A close is not necessarily complete when this function returns. The application may need to wait for the notify function to be called with the IRCOMME_CLOSE event. |

When the application is finished sending or receiving data and/or status information, then IrcClose() can be called. This function will disconnect IrLAP, and any pending data (data not sent by the driver or read by the application) will be lost. Thus, the application must make sure that no pending data exists before calling IrcClose().

The way to check for pending data is to keep track of IrcWrite() calls and the corresponding calls to the notify function with the EVT_IRD_IRCOMME_WRITE event.

For each call to IrcWrite(), there is a corresponding event sent to notify the application. When all IrcWrite() calls have been confirmed with notification events, then it is known that all data has been sent (no pending data).

Please note that the IrCOMM driver sends an initial event to the user application with the EVT_IRD_IRCOMME_WRITE event when a connection first comes up. This initial call to the notify function is not the result of an IrcWrite() call.

When IrcClose() returns, the IrLAP link is not necessarily disconnected. If IrcClose() returns ERR_IRC_STATUS_PENDING, then IrLAP is not disconnected; the application should wait for IrCOMM to notify the user application with the EVT_IRD_IRCOMME_CLOSE event, signaling that the close is complete (IrLAP is disconnected).

## OBEX Application Layer APIs

The OBEX APIs, shown in Table 19.5, are defined to operate either in Client or Server mode. To simplify the use of IrOBEX, Client mode is configured to do a “Put” operation only, and Server mode is configured to respond to a “Put” request by a peer IrOBEX client. This mode of operation is fixed and cannot be changed by the user. The Client mode of IrOBEX handles all Put requests started by the user, while the Server handles all Put requests initiated by the peer device in the background.

When ObxInit() is called, both the IrOBEX Client and Server services are started and ready to send and / or receive “Put” requests.

The IrOBEX service entity has a concept of Inbox and Outbox. When a “Put” operation is initiated successfully by a peer IrOBEX device, the IrOBEX Server will store the Content of the “Put” operation in the Inbox and notify the user. The size of the Inbox is subjected to a maximum of 2 Kbyte of data. Anything more than that is rejected automatically. The notification to users is event driven; the Server will send the event EVT_IRD_OBSE_RX_COMPLETE.

With this, there is a series of APIs that the user can use to check the length of the IrOBEX data and retrieve it into the user’s own buffer storage.

To start a “Put” operation, the user must first fill the Outbox with user data through the APIs provided. After that, ObxPut() is called to start the operation.

### Starting and Stopping IrOBEX Services

Before starting the IrOBEX services, IrDA transport would have to be started by calling IrdInit(). IrOBEX Client and Server Services are then started by a simple call to ObxInit() with the user’s TASK_ID as a parameter. Callbacks to the user occur through a series of events such as the other IrDA management services sent by the IrOBEX client and server.

Conversely, stopping the IrOBEX services is accomplished with a call to ObxDeinit().

### Initiating a “PUT” operation

When a user intends to use the IrOBEX services to send an Object (for example, a Vcard file) to an IrOBEX-capable device, the following steps are taken:

- Initiate an IrDA TinyTP connection if it is not already started.
- Initiate an IrOBEX connection to the peer's IrOBEX Server.
- Build the header to the required format in Unicode.
- Save the pending data into the Outbox.
- Start the “Put” operation.
- Disconnect from peer's IrOBEX Server.
- Disconnect IrDA TinyTP if necessary .

### Receiving an Object and retrieving object from Inbox

After IrOBEX services have been initialized, the IrOBEX server is ready to receive any Object up to a size of 2 Kbyte. A “Put” operation would have to be requested by the peer IrOBEX. The IrOBEX Server will store the Content of the “Put” operation in the Inbox and notify the user. The Server will send the event EVT_IRD_OBSE_RX_COMPLETE to the user task. Inside the user's task, the routine must wait on the event been generated.

To retrieve the Object from the Inbox, users can use the APIs provided Table 19.5 to copy the content into the user buffer.

**Table 19.5 OBEX APIs**

| OBEX API | Description |
|---|---|
| STATUS **ObxInit** (TASK_ID AppCallback) | Initialize the OBEX Client and Server. |
| STATUS **ObxDeinit**(void) | De-initialize the OBEX Client / Server. |
| STATUS **ObxSaveInbox** (P_U8 buff, U16 len) | Save Inbox contents into a buff provided by the user with the specified length, len. |
| STATUS **ObxGetInboxLen** (void ) | Get the length of the Inbox body. |
| STATUS **ObxSaveName** (P_U8 buff, U8 len) | Save Name from the Inbox to the given object store item. |
| U8 **ObxGetNameLen** (void) | Get the length of the Inbox Name. |
| STATUS **ObxPutOutbox** (P_U8 buff, U16 len) | Put data into the Outbox to be sent. |
| void **ObxAbort**() | Abort the current client operation. No check is made to see if an actual operation is in progress. This function will cause an IrOBEX Abort packet to be sent to the Server if an operation is in progress. |
| STATUS **ObxConnect**(void) | Perform the IrOBEX connect operation. ObxConReq() must be executed first to establish a TinyTP connection. |
| STATUS **ObxConReq**(void) | Start a TinyTP connection to an IrOBEX device. Attempt to set up a TinyTP connection to an IrOBEX Server. Completion of this request is reported by a call to the application's client callback function with the appropriate event. |

| OBEX API | Description |
|---|---|
| STATUS **ObxClientDisconnect**(void) | Disconnect the client IrOBEX connection. |
| STATUS **ObxDiscReq**(BOOL Force) | Disconnect the client's TinyTP connection. If a server connection exists, then this function will not disconnect IrLAP unless Force is equal to TRUE. |
| STATUS **ObxPut**(void) | Issues an IrOBEX PUT over Connected transports and sends Contents in Outbox over. |
| ObxAbortReason **ObxGetAbortReason**(void) | Obex Session has aborted; get the Reason Code for abortion. This call is only valid during calls to the client application callback with the event OBCE_ABORTED. |
| BOOL **ObxHeaderBuildUnicode** (ObxHeaderType Type, U8* Value, U16 Len) | The function builds client-specific versions of the generic IrOBEX header. |
| BOOL **ObxHeaderBuild4Byte** (ObxHeaderType Type, U32 Value) | The function creates a 4-byte style header for transmission with the client's request. Four-byte headers have the seventh and eighth bits in their type set to one. |

## Summary

The IrDA management services support IrCOMM and IrOBEX implementation for the application. IrCOMM services provide serial and parallel port emulation to the applications. Through IrCOMM services, the applications can communicate with the outside world over an IrDA infrared link instead of the wired link.

The IrOBEX services provide the ability to Put data objects very simply and flexibly, thereby enabling rapid application development and interaction with a broad class of devices including PCs, PDAs, data collectors, and cameras.

IrOBEX services make it extremely simple for one or more applications within a device to send or receive data objects. The data objects could be files, images, records, or any unit of data that makes sense to the applications involved. Applications can avoid the usual complex programming tasks associated with IR—performing discovery, making connections and managing transactions. Instead they make a simple call to send (as if dropping the object in the mail), and they are notified when a complete incoming object is ready to process (as if it had arrived in the mail).

## Code Examples

**Listing 19.1 Opening IrCOMM port with IRCOMM_Open()**

```
      appState == APP_STATE_INIT;
/* Application is initializing */

      if (IrcOpen(gIrcTaskId) == ERR_IRCM_STATUS_FAILED)
      {
        printf("Fatal Error: Unable to bind to the stack\n");
        return FALSE;
      }


      event = EvtGet();

      switch (appState)
      {
        case APP_STATE_INIT:
        if (event == EVT_IRD_IRCOMME_WRITE)
        {
  /* The driver is now open and user could add custom code here*/

          appState == APP_STATE_IDLE;
          }
          break;

          :
```

```
        :
        :
    }
```

**Listing 19.2 Writing Data with IRCOMM_Write()**

```
U8 writeBuff[MAX_WRITE_BUFF_SIZE];
U16 writeLen;

   if (IrcWrite(writeBuff, writeLen) != ERR_IRC_STATUS_FAILED)
   {
      /* The write was successful */
      printf("Bytes written is %10ld.\n", writeLen);
   } else {
      /* The write failed */
      printf("Could not write data.\n");
   }
}
```

**Listing 19.3 Reading Data with IRCOMM_read()**

```
U8 readBuff[MAX_READ_BUFF_SIZE];
   U16 len;

   /* Handle any incoming data */
   if (event == EVT_IRD_IRCOMME_READ) {
      while (len = IrcRead(readBuff, MAX_READ_BUFF_SIZE))
      {
      /* Data has been read, print it out on the screen */
      readBuff[len] = 0;
      printf("%s", readBuff);
   }
}
```

**Listing 19.4 Closing the IrCOMM port with IRCOMM_Close()**

```
if (IRCOMM_Close() == IRCOMM_STATUS_SUCCESS) {
    /* IrCOMM is closed */
    printf("IrCOMM closed."\n);
} else {
    /* IrCOMM close is pending, must be handled in callback. */
    appState = APP_STATE_CLOSING;
    printf("IrCOMM closingÖ"\n):
}

/*In cases where IRCOMM_Close() returns IRCOMM_STATUS_PENDING */
/* the notify function should handle the IRCOMME_CLOSE event. */
/* The application's notify function */

if (appState == APP_STATE_CLOSING)
      {
        /* Application is closing */
        if (event == EVT_IRD_IRCOMME_CLOSE)
        {
            /* The driver is now closed */
            appState == APP_STATE_CLOSED;
        }
    }
```

**Listing 19.5 Starting IrObex Services with Obx_Init()**

```
/* Task Id for User process */
TASK_ID gObxTaskId;
STATUS status;
:
:
status = ObxInit(gObxTaskId);
if (status != ERR_OBX_STATUS_SUCCESS )
{
  printf("APP_Init Error!\n");
}
else
{
  printf("IrObex started successful\n");
    /* Add the device info and hints */
IrdSetDeviceInfo((U8*)deviceInfo, (U8)deviceInfoLen);
}
```

**Listing 19.6 Initiating TinyTP connection and IrObex connection to peer IrObex server**

```
STATUS status;
:
:
status = ObxConReq();
if (status == ERR_OBX_STATUS_MEDIA_BUSY)
{
  printf(" Media Busy, try later.\n");
}
else if (status == ERR_OBX_STATUS_SUCCESS)
{
clientState = CLIENT_OBEX_CONNECT;
  printf("Already connected at TTP\n");
}

if (clientState == CLIENT_OBEX_CONNECT)
{
/* Send the OBEX Connect request. */
status = ObxConnect();
if (status != ERR_OBX_STATUS_PENDING)
{
            printf("Client: OBEX Connect request failed to 
start.\n");
            clientState = CLIENT_DISCONNECT;
:
:
/* Do Disconnect routine*/
}
}
```

**Listing 19.7 Disconnecting from TinyTP connection and IrObex connection**

```
if (clientState == CLIENT_OBEX_DISCONNECT)
{
if (ObxDisconnect() == ERR_OBX_STATUS_PENDING)
{
  /*Obex disconnect pending wait for event*/
            return;
        }
}
:
:
:
if (clientState == CLIENT_DISCONNECT)
{
if (ObxDiscReq(FALSE) == ERR_OBX_STATUS_FAILED)
        printf("Disconnect request failed\n");
}
```

**Listing 19.8 Starting a "Put" Operation**

```
/* Unicode is 16-bit while Name is 8-bit char; make sure there is
enough buffer to use but not more than OBS_MAX_NAME_LEN */
U8            ucName[(128+1)*2];
U8            nameLen;
STATUS      status;
:
:
nameLen = sizeof(vcard_name) * 2;
/* Now convert vcard_name into UNICODE and store into ucName */
ReadNameUnicode((char *)ucName, nameLen);

/* Create Name for the Object in the Header*/
ObxHeaderBuildUnicode( OBXH_NAME, (U8 *)ucName,  nameLen);

/* Create Length header (only when connected) */
ObxHeaderBuild4Byte(OBXH_LENGTH, sizeof(MyVCard));

/* Store into Outbox before calling ObxPut()*/
status = ObxPutOutbox(&MyVCard, sizeof( MyVCard ));
if (status == ERR_OBX_STATUS_FAILED)
{
  printf("Size too big for Outbox\n");
  clientState = CLIENT_OBEX_DISCONNECT;
  DoDisconnect();
}

printf("Client: Starting Put operation.\n");

/* Call Put function. Put will send object stored in Outbox over*/
```

```
status = ObxPut( );

if (status != ERR_OBX_STATUS_PENDING)
{
  if (status == ERR_OBX_STATUS_MEDIA_BUSY)
  {
            printf("Client: Put function failed, Media Busy!\n");
        }
        else
        {
            printf("Client: Put function failed, reason %d.\n",
status);
        }
}
else if (status != ERR_OBX_STATUS_SUCCESS)
(
  printf("Put Successful");
)
```

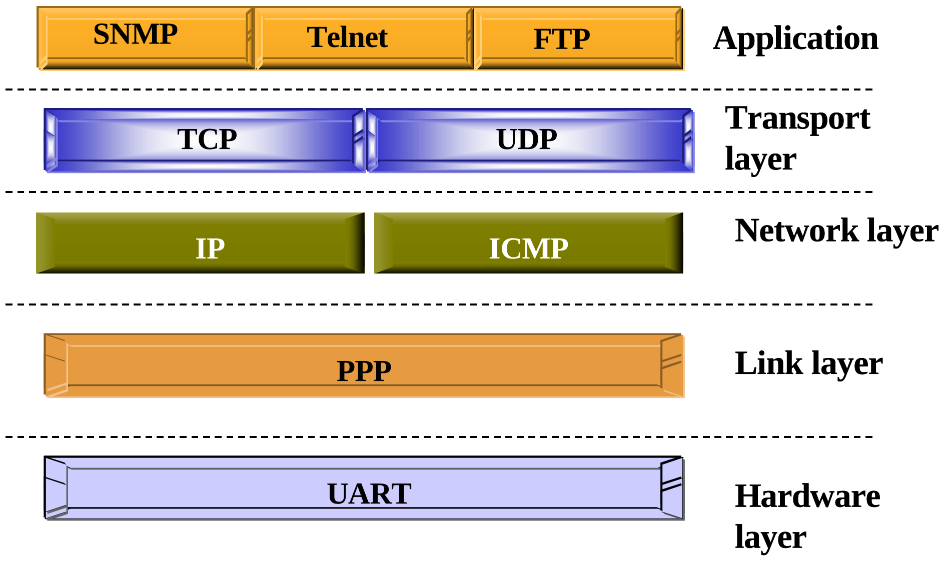
20

# Networking Services

Networking or TCP/IP services support a set of protocols to allow cooperating computers or devices to share resources across a network. Protocols such as IP, TCP, and UDP provide “low-level” functions needed by applications that need to preform ftp, telnet, and e-mail functions.

Figure 20.1 shows the protocols from link/physical layer to application layer. To perform any networking activities in the PPSM-GT environment, developers need to be familiar with networking and used the APIs provided for handling the transport layers and application layers. Accessing the network and physical layers are restricted to network configuration, and the rest of the programming is handled by the system networking services.

**Figure 20.1 Networking Protocol Structure**



This chapter is organized into the following main sections:

## Networking Fundamentals

The networking services allow diverse systems to communicate with each other. Typically, the communication is between the target embedded system and a server. The target application interfaces with the outside world, performing some form of data collection through easy-to-use APIs such as readsocket(). When necessary, the target application opens a connection to the server and transmits the data using socket() and writesocket(). The networking services take on the responsibility of providing a reliable connection and reliable data transport when using TCP/IP.

Table 20.1 shows the networking protocols that are supported in PPSM-GT. Only brief descriptions of the protocols are provided; a detailed description of each protocol is beyond the scope of this document.

**Table 20.1 Networking Protocols Supported**

| Protocol | Description |
|---|---|
| TCP | Transmission Control Protocol: Transport layer with connections, flow control and error correction.<br>Provides a reliable character stream, as opposed to IP. Able to recover lost segments, and uses acknowledgment with retransmission for resolving transmission errors. |
| UDP | User Datagram Protocol: Simple connectionless transport layer.<br>Best effort delivery, includes checksum and allows broadcast. |

| Protocol | Description |
|---|---|
| IP | Internet Protocol: The network layer. Unreliable, datagrams may be lost, duplicated or delivered out of order. Provides a universal data header, independent of a network technology. |
| ICMP | Internet Control Message Protocol: part of IP for practical purposes. Ping uses an ICMP “echo request” and “echo reply.” |

## Types of Networking Services

There are two main groups of networking services provided with PPSM-GT. These services are designed to reduce developers’ loading on the networking details and focus on the link configuration and usage. These two groups of services complement one another such that both need each other to create meaningful networking activities. The order in using the two services are important and will be discussed in later section.

The two group of services are

- Link Setup Services
- Transport/Socket Services

### Link Setup Services

Link setup services are services that handle network connections. The protocol involved is the Point to Point Protocol, or PPP. The PPP is used to establish a link to a single remote host. This is commonly used in data acquisition, internet connectivity and other arenas. The link setup services setup the foundation for the networking activities that is in order for any networking activities to take place the link setup services need to be called first.

Link setup services is organized into:

- “Link Configuration”
- “Server Configuration”

### Link Configuration

This version of PPSM-GT networking services does not support direct link establishment and support only Hayes-compatible modems.

There are two ways to establish a link: direct and through modem dialup. This version of PPSM-GT does not support direct link establishment. The modem dialup script provided is for Hayes-compatible modems only and used to establish a point-to-point connection with another host. When using a modem to establish the connection, the modem dialup script take cares of link establishment and responds to queries from a terminal session before the PPP handshake can begin.

There are several dialing parameters on the modem script that are user specified and need to be provided by the user. The parameters, which appear in the following list, are modified through the APIs provided. The following shows the modem setup APIs, and details of the APIs can be found in "Programming with Link Setup Services." :

- NetConfigPPP( ) - It configures the user ID and password required for a PPP connection. Usually these are the same as those used in NetConfigISPAccount for a dial-up connection.
- NetConfigModem( ) for setting the communication baud_rate with the modem, the host name and the comm port. Configures the low level driver before the stack is started.

**NOTE** There is programming sequence that need to be observed especially when initialing the networking stack. Please refer to Programming with Link Setup Services for details.

### Server Configuration

In server configuration, there are APIs for configuring the DNS, ISP and local host server. By default PPSM-GT networking services set the servers to its' default values. If the user did not set any preferred server using the NetConfigDNS() API, the system will use the default values.

The following are the APIs for server setup:

- NetConfigDNS() - It configures both the primary and secondary DNS server's IPs. There are two default DNS server and this API will change the specified server's IP address to new IP Address.
- NetConfigGateway() - It sets the ISP server's (gateway) IP. A value of 0.0.0.0 indicates an IP address will be obtained form the ISP; 0.0.0.0 is the default value.
- NetConfigLocalHostIP() - It sets the local host's IP. A value of 0.0.0.0 indicates an IP address will be negotiated with the ISP; 0.0.0.0 is the default value
- NetConfigISPAccount( ) - It configures the ISP account's dial-up number, the user ID, and the password.
- NetConfigMachineName() - It sets the name of the local host. The default value is "none".

## Transport/Socket Services

The transport or socket services handle the transportation of information from one end of the network connection to the other end. They concern the quality and not the content of the information.

The transport services provide a generic interface to communications protocols through communication end points known as sockets. PPSM-GT sockets are based on Berkeley Software Distribution (BSD) sockets from the University of California at Berkeley. They provide a generic interface to network level communications protocols for common operations involved in network computing. These include sending data, receiving data, and establishing connections.

To minimize confusion in socket programming, the PPSM-GT socket APIs used the same naming convention as that of the BSD socket APIs as much as possible.

**NOTE** Not the full set of socket is supported. PPSM-GT socket services consist only a subset of the unix socket API. The naming of the API is kept similar to that of the BSD socket API to minimize user confusion.

The protocol supported in this layer is TCP, UDP and IP and offers the following functions:

- Provide access to communications networks such as the Internet.

- Enable communication between unrelated processes residing locally on a single host computer and residing remotely on multiple host machines.
- Sockets provide a sufficiently general interface to allow network-based applications to be constructed independently of the underlying communication facilities. They also support the construction of distributed programs built on top of communication primitives.
- The socket APIs are the application program interface for Transmission Control Protocol/Internet Protocol (TCP/IP).

The Berkeley sockets model conceptualizes network communications as taking place between two endpoints, or sockets. Analogies have been drawn that compare plugging an application into a network to plugging a handset into the telephone system, or an appliance into a electrical system. Most sockets programs — under Unix or Windows — utilize a client/server approach to communications.

Before looking at the concept of the client and server model, let look at the limitation of PPSM-GT sockets as compare to the unix socket.

### PPSM-GT sockets versus UNIX standard sockets

The PPSM-GT sockets are only approximate the many UNIX functions interacting with sockets, and only a subset of the socket APIs are supported. The socket implementation is limited to networking only. Unlike UNIX's version where files and even other things can be mixed together using the same set of functions, the PPSM-GT socket APIs read, write, select, close and fcntl functions are limited to socket operations only. Therefore to differential these functions with the generic UNIX function names (e.g. fcntl(), select(), read(), write(), close()), the word “socket” is appended after the function names. For example read() becomes readsocket() etc.

The following are the PPSM-GT sockets limitations as compared to UNIX sockets:

1. The UNIX sockets are really an intertask communication system, not a networking interface. They can be used to map to the various UNIX file systems, and they can mix files and sockets and even other things in one operation.
2. The use of the functions ***fcntl()***, ***select()***, ***read()***, ***write()***, and ***close()*** for networking purposes will easily cause conflicts. PPSM-GT

networking services change these names by adding“***socket***” to them.

3. The UNIX sockets have an interface to the UNIX signals, which again have an interface to just about any UNIX function.
4. Some BSD socket features are implicitly not re-entrant. These include the function ***gethostbyname()*** and all uses of *errno*. This is, of course, more a multitasking question than a networking question.

## The Client/Server Model

.Rather than trying to start two network applications simultaneously, one application is theoretically always available (the server) and another requests services as needed (the client).

The server creates a socket, "names" it so that it can be identified and found by a client, and then listens for service requests. A client application creates a socket, finds a server socket by name or address, and then "plugs in" to initiate a conversation. Once a conversation is initiated, data can be sent in either direction. The client can send data to and receive data from the server, and the server can send data to and receive data from the client.

The specifics of the conversation are unique to each set of client and server applications. Both applications must know what messages and data to expect from the other, and both must follow some mutual rules about when to send and when to expect to receive data. In order to successfully communicate, both the client and the server must speak the same language—they must both use the same protocols (like HTTP, POP, FTP, etc.).

Bear in mind that the concept of a socket is purely an abstraction used in the PPSM-GT Networking API. It’s not necessary that client and server applications both be written with PPSM-GT Networking APIs in order to communicate. A program written with the networking API can communicate with many different types of systems, as long as they use the same protocols. For example, a Web Browser written with PPSM-GT can retrieve files from a Web Server written for a different operating system in a different language with a different network interface. As long as both programs "speak" the same transport protocol (like TCP/IP) and the same application protocol (like HTTP) then they can communicate.

## Socket programming model

Sockets are based on the client/server model. The behavior and the programming differ between them. Figure 20.2 and Figure 20.3 provide a programming references for the server and the client under both connection and connectionless system. The following sections provide the programming stages to enable socket networking, and it would be beneficial to reference the figures during the description of the following stages:

- Creating Sockets
- Naming a socket
- Accepting and Making Socket Connections
- Transferring Data
- Shutting Down Socket Operations
- Translating Network Addresses

### *Creating Sockets*

A socket is created with the socket() API. This API creates a socket of a specified domain, type, and protocol. Sockets have different qualities depending on these specifications.

- A communication domain indicates the protocol families to be used with the created socket.
- The socket type defines its communication properties, such as reliability, ordering, and the prevention of duplicating messages.
- Some protocol families have multiple protocols that support one type of service. To supply a protocol in the creation of a socket, the programmer must understand the protocol family well enough to know the type of service each protocol supplies.

In PPSM-GT, when creating a socket, the following are the options when creating a socket:

- Domain: PF_INET for TCP/IP
- Type: TCP/IP, UDP/IP or raw (See “Selecting socket type”)
- Protocol: Only TCP/IP and related are available; always enter 0 (zero) to have protocol auto-assigned based on type and domain

This function returns a socket handle or -1 if there is an error.

### *Selecting socket type*

Two basic kind of connections are available:

- Stream socket or TCP
- Datagram socket or UDP

The primary differences are:

1. TCP performs error correction and flow control, and UDP does not.

   Developers can read TCP like a local disk file. When reading TCP, developers should check for errors, and as errors occur quitting reading is recommended. Reading UDP is different; there is no error correction. Error checking with UDP would be difficult, which must be taken in account when writing an application using UDP. It is best to leave UDP for pre-written applications, such as TFTP and BOOTP.

2. UDP is a packet protocol, and TCP is a byte-stream protocol.

   With TCP, the number of bytes read could not be predicted with certainty, and the given amount of data needs to return.

   TCP is designed as a byte-stream protocol. This byte-stream nature explains why higher level protocols are needed on top of TCP. Higher level protocols such as FTP, SMTP (e-mail), and HTTP (Web) always have a data size in a known place in the header, thereby telling the application how much data to read.

   So, when using TCP without any higher level protocol, developers need to know how much data to expect and design it in the application. A header could be included on top of the data that tells the remote host how much data will be sent.

   With TCP, the application may write 10 times with 100 bytes of data each time. TCP takes that data and streams it. So, the remote application may get one TCP segment with 1000 bytes, two segments with 500 bytes each, or some other distribution.

### *Naming a socket*

Naming a socket normally applicable only for server application and it optional. It serves as a friendly identification of the socket for the client application during selection. An application can bind a name to a socket using bind(). The socket names used by most applications are readable strings. However, the name for a socket that is used within a communication domain is usually a low-level address. The form and meaning of socket addresses are dependent on the communication domain in which the socket is created. The socket name is specified by a sockaddr structure.

### *Accepting and Making Socket Connections*

Once a socket is made, depending on whether the application is a a server or client, they behave differently. To facilitate discussion the following section is organized into the client behavior and server behavior:

### *Client Behavior*

In the client application after the socket is created, it need to connect to a server. The server is identified by an address/port pair. So the client application need to fill a sockaddr_in structure with the right info and call connect(), which returns a -1 if an error occurs. For a better understanding, please refer to the code example in Example 20.3.

A port number is a 16-bit unsigned number (except port 0 is not used). Certain commonly used ports on machines are given symbolic names.A port can be any number between 1000 and 64000. The TCP/IP socket port that the server listens to for the receiving and sending of requests. Port 80 is typically used by HTTP servers and browsers.

### *Server Behavior*

The server behaves quite differently after the socket is created. For the server, it needs to be bound to a port. This is done using the bind() function. It takes the same parameters as connect(); the family, port, and address. The family is the same, and the port is the same, the address is INADDR_ANY. And it will set up a port with the given number and it will be bound to the socket.

Then the listen queue is established with listen(). The first parameter is the socket handle and the second is the size of the listen queue.

It will accept a socket connection by accept() when it received a connection required from a client. The accept() call is used by a server application to perform a passive open for a socket. The socket will remain in the LISTEN state until a client establishes a connection with the port offered by the server. The return value from this function is an identifier for a newly created socket over which communication with the remote client can occur. The original socket remains in the LISTEN state, and can be used in a subsequent call to accept() to provide additional connections. The following illustrate the steps taken during a connection requests at the server:

- When binding to (0,0), it will assigned a predefined port no (say 9999) and bind it (same as binding (0, 9999), instead of randomly assigned a port no.
- Do a listen(0,0) --> This shall put socket(0,9999) in the LISTENING State. Then do an accept().
- When a SYN segment comes, it will get a listening socket from port 9999 instead of the normal get_sock().
- From there, it will create its own socket filled with the source and destination IP & port no, and insert it into its proper place based on its port no.

### Type of Networking Connection

The client will make a socket connection request through connect(). Stream sockets (TCP/IP sockets) need to be connected before use, whereas other types of sockets, such as datagram sockets (UDP/IP sockets), need not establish connections before use.

There are two types of connections available when making a socket connection. Figure 20.2 shows the socket APIs involved in a stream/ connection-oriented system, and Figure 20.3 shows the socket APIs involved in a peer/connectionless-oriented system.

**Figure 20.2 APIs Used in a Connection-Oriented System**

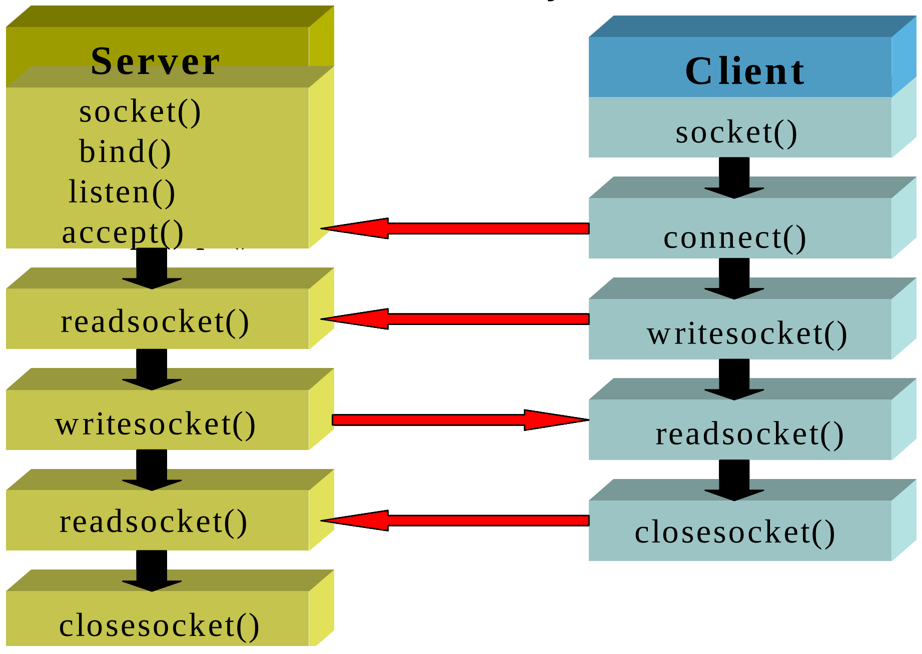
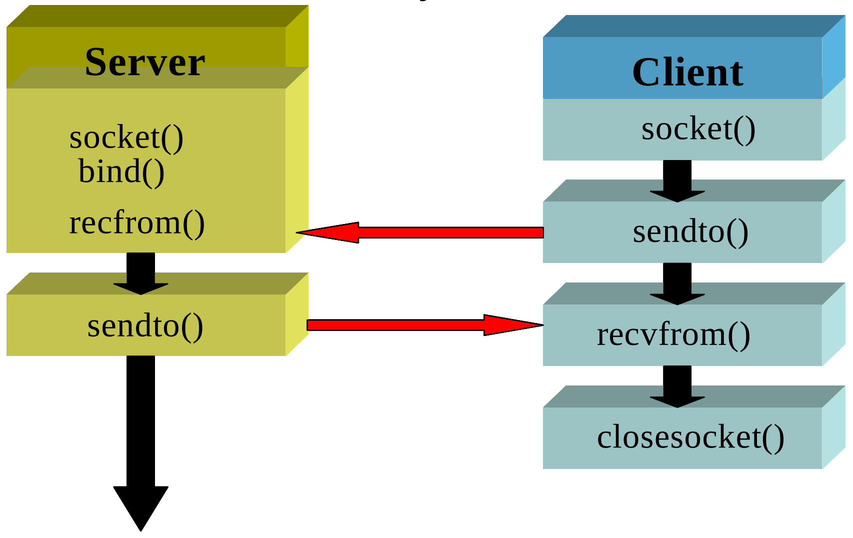


**Figure 20.3 APIs Used in a Connectionless System**



### *Transferring Data*

Sockets include a variety of calls for sending and receiving data. readsocket() and writesocket() can be used on sockets that are in a connected state, as in Figure 20.2. recv() and recvfrom() permit callers to specify or receive the address of the peer socket, as in Figure 20.3. These calls are useful for connectionless sockets, in which the peer sockets can vary on each message transmitted or received. The sendmsg() and recvmsg() APIs support the full interface to the IPC facilities. Besides offering scatter-gather operations, these calls allow an address to be specified or received and support flag options.

writesocket() takes as arguments the socket identifier, a pointer to a buffer containing the data and the size of the data.

readsocket() also takes socket identifier that indicates a socket, and like writesocket() require a connected socket since no destination is specified in the parameters of the system call.

For connectionless connection, to send datagrams, one must be allowed to specify the destination. The call sendto() takes a destination address as an argument and is therefore used for sending datagrams. The call recvfrom() is often used to read datagrams, since this call returns the address of the sender, if it is available, along with the data.

Finally, there are a pair of calls that allow the sending and receiving of messages from multiple buffers, when the address of the recipient must be specified. These are sendmsg`()` and recvm`sg()`. These calls are actually quite general and have other uses, including, in the UNIX domain, the transmission of a file descriptor from one process to another.

### *Shutting Down Socket Operations*

Once sockets are no longer in use, they can be closed or shut down using the shutdown() or closesocket() API.

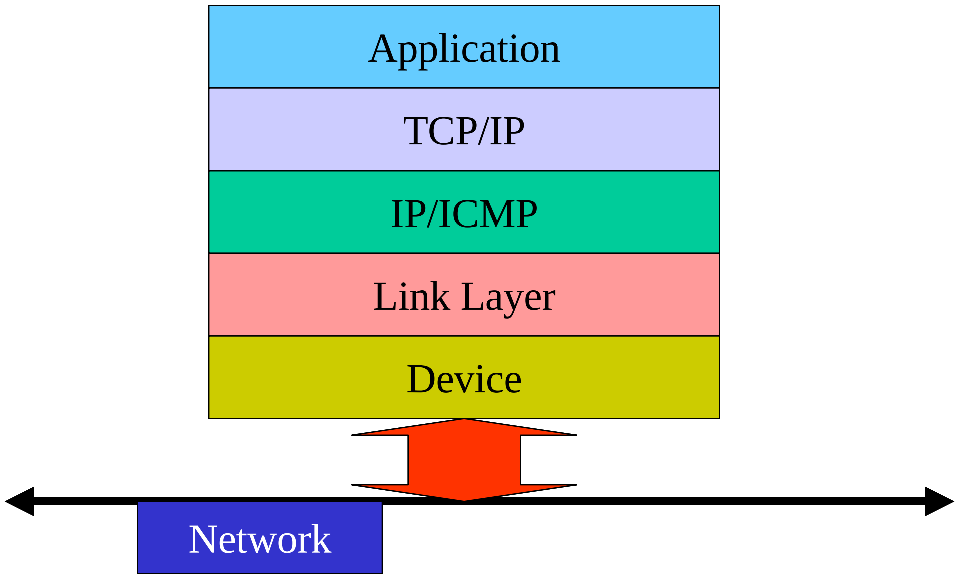
### *Translating Network Addresses*

Application programs need to locate and construct network addresses when conducting the interprocess communication. The socket facilities include subroutines to:

- Map addresses to host names and back
- Map network names to numbers and back
- Extract network, host, service, and protocol names
- Convert between varying length byte quantities
- Resolve domain names

## Programming with Networking Services

**Figure 20.4 Programming model**



The logical relationships between the protocols are illustrated in the programming model shown in Figure 20.4. Programming with the networking services follows the same logical relationship except that most of the programming is already done. To use the networking services, the work is simplified to configuring the network and using the networking services like a black box. These operations can be as simple as opening the connection, writing or reading information and closing the connection, in some cases.

Programming the networking services is divided into the following categories:

- Programming with Link Setup Services
- Programming with Transport/Socket Services
- Programming the Networking task

## Programming with Link Setup Services

As mentioned in Link Setup Services, there are two group of link setup APIs: the link configuration and server configuration APIs. The link configuration APIs as shown in Table 20.2 are used for setting up the network configuration and network initialization. To initialize the network, first set up the network configuration with the APIs such as NetConfigPPP() and NetConfigModem(). Then initialize the networking stack to that configuration with NetInit(). The sequence of programming is important and must be observed.

As shown in the code example, "Networking API calling sequence.", the networking parameters need to be set up first using NetConfigPPP() to configure the IP address of the server address and NetConfigModem() to configure the modem parameters before caling NetInit to save the configuration onto the networking stack.

**NOTE** All NetConfigXXX APIs need to be called before NetInit( ) to be effective. These are for items whose values are read (and often only read) during the initialization process.

Once, the NetInit() is called, calling the NetConfigModem(), or other NetConfigXXX() will not change the networking configuration until the network is terminated and start up again.

Normally the network needs to be configured once for the same network connection. For different network connections, where new IP address is required the PPSM-GT networking stack has to be reconfigured to new configurations and this included shutting down the network with NetDeinit() and setup the configuration using NetConfigPPP(), then called NetInit() to initialize the networking stack.

**Table 20.2 Link Configuration APIs**

| | |
|---|---|
| VOID **NetConfigDNS**(P_S8 ipAddress, U8 which) | It sets the DNS server's IP. There are two default DNS server and this API will change the server's IP address to ipAddress specified by the parameter which. |
| VOID **NetConfigISPAccount**(P_S8 UserID, P_S8 Passoword, P_U8 PhoneNum) | It configures the ISP account's dial-up number, the user ID, and the password |
| VOID **NetConfigGateway**(P_S8 ipAddress) | It sets the ISP server's (gateway) IP. A value of 0.0.0.0 indicates an IP address will be obtained form the ISP; 0.0.0.0 is the default value |
| VOID **NetConfigLocalHostIP**(P_S8 ipAddress) | It sets the local host's IP. A value of 0.0.0.0 indicates an IP address will be negotiated with the ISP; 0.0.0.0 is the default value |
| VOID **NetConfigMachineName**(P_S8 Name) | It sets the name of the local host, and the name cannot be longer than or equal to 9 characters. The default value is "none". |
| VOID **NetConfigModem**(P_S8 comPort, NET_DRIVER comDriver, P_S8 baudRate) | Configure the modem port which is the low level driver before the stack is started. |
| VOID **NetConfigPPP**(P_S8 UserID, P_S8 Password) | It configures the user ID and password required for a PPP connection. Usually these are the same as those used in NetConfigISPAccount for a dial-up connection. |
| STATUS **NetDNSResolve**(P_S8 Fullname, IP_ADDR* iidp) | Resolves a domain name to an IP address or vice versa |

| | |
|---|---|
| STATUS **NetInit**(void) | Performs general initialization, such as initialization of tables and buffers. It must be the first network function called and cannot be called again unless the function ***NetDeinit()*** has been called first. |
| VOID **NetDeinit**(void) | It shuts down any open port and terminates the related driver task(s) |

## Programming with Transport/Socket Services

Programming with socket services follows the same networking programming model described in Figure 20.4. To be able to do any socket programming, the link layer has first needed to be configured and set up. In the other word, before any programming of the socket service can take place, the programming of link layer has to be done first.

Many of the BSD socket routines use a pointer to structure sockaddr, which specifies network address information. The sockaddr structure is a generic structure that can be used with a number of different communications protocols.

PPSM-GT sockets only use the Internet Protocol (IP) and therefore only require the use of the Internet structure sockaddr_in. Values are assigned to sockaddr_in and passed into the socket routine via the sockaddr parameter. This requires a typecast to sockaddr *. The discussion of the ***connect()*** function provides an example. Here are the structure definitions:

```
struct sockaddr {                    /* generic socket address */
        unsigned short sa_family;    /* address family */
        char sa_data[14];            /* up to 14 bytes of address */
};
```

In practice, this is used almost as a void pointer. The true Internet address structure is:

```
struct in_addr {                     /* Internet address */
        unsigned long S_addr;
};


struct sockaddr_in {                 /* Internet socket address */
        short sin_family;
        unsigned short sin_port;
        struct in_addr sin_addr;
        char sin_zero[8];
};
```

### PPSM-GT Socket APIs

The PPSM-GT Sockets provides the function calls identified in Table 20.3.

**Table 20.3 Socket APIs**

| Socket API | Description |
|---|---|
| accept | Accepts the next incoming connection on a passive (listening) socket. Returns with a new socket. In other words, this acts as a "socket fork" function. For streaming sockets only (i.e. TCP) |
| bind | Binds a name to a socket. |
| closesocket | Closes a connection on a socket. Flushes streams, calls low-level close, releases buffers |
| connect | Initiates a connection on a socket. Used to specify the remote endpoint address of for sockets |
| socket | Creates a socket and returns a socket ID number (descriptor) for that socket SOCK_STREAM: TCP/IP SOCK_DGRAM: UDP/IP<br>Others: none (i.e. SOCK_RAW is NOT supported) |
| fcntlsocket | Controls socket flags. Allows a socket to be set to use non-blocking semantics, and also allows the current setting to be retrieved |
| gethostbyaddr_r | Returns host information when an IP address is supplied |
| gethostbyname_r | Returns the IP address that corresponds to a host name. |
| getpeername | Returns the address of the peer connected to socket s inside the peer data structure. Useful for connections obtained using accept( ) to find out the remote address |

| Socket API | Description |
|---|---|
| getsockname | Returns the local address and protocol port number of the specified socket s inside data structure name. On return, nameLen contains the actual size of the name returned in bytes. |
| getsockopt | gets options on sockets. |
| htonl | Converts an unsigned long interger (U32) from the local byte order to the network byte order |
| htons | Converts an unsigned long interger (U16) from the local byte order to the network byte order |
| listen | Puts socket s into LISTEN mode, which passively waits for connection requests |
| inet_addr | Converts an IP address in dotted decimal form to the internal binary form that sockets functions expect |
| inet_ntoa | Converts an IP address to dotted decimal form from the internal binary form that sockets functions expect |
| ioctlsocket | sets control parameters for a socket. |
| ntohl | Converts an unsigned long integer (U32) from the network byte order to the local byte order |
| ntohs | Converts an unsigned long integer (U16) from the network byte order to the local byte order |
| readsocket | receives a message from a socket ID. |
| recv | receives a message. |
| recvfrom | Get a message from the queued messages on the connection. Store the sender’s address in the from structure. |
| recvmsg | establishes a connection and receives a message. Receives and stores a message according to information inside msg |
| selectsocket | Waits for activity on a set of sockets. The function returns when any of the specified conditions occur or when the time out period expires |

| Socket API | Description |
|---|---|
| send | sends a message on an established connection. Transmits buf to a remote host connected to the local sockets. |
| sendmsg | sends a message that can be split between buffers. |
| sendto | Sends a message to a remote host specified by the argument 'to' |
| setsockopt | sets options on sockets (described with getsockopt). |
| shutdown | Terminates transmission, reception, or both on a socket |
| writesocket | Transmits buffer to a remote host connected to the local sockets |

### Programming the Networking task

The networking task is the task that normally create to run the networking functions. The following are guidelines for creating he networking task.

1. The networking task should have a priority higher than that of the Sci_task to avoid the KnlCreateTask error.
2. If the application framework is used, the framework task should have a priority that is lower than that of the Sci_task to avoid the modem connection problem.

## Summary

The PPSM-GT networking services provide APIs to program networking activities at the link, transport and application layers. Network configurations such as IP address, connection speed and other configurations are performed through the APIs.

The PPSM-GT sockets kept the same naming convention as that of the BSD socket APIs to avoid un necessary confusion for networking socket programmer. The socket services behave just like the BSD socket API to enable programmer to provide a generic interface to network level communications protocols for common operations involved in network computing. These include sending data, receiving data, establishing connections, and configuring network protocols.

## Code Examples

**Listing 20.1 Examples for initialize all network interface**

```
/* initialize all network interfaces */
main()
{
/* Setup the low level driver/port, which must be set before the
"NetInit()".*/
    NetConfigModem(UART_2, PPSMSCI, "115200");

/* If the PPP required, setup the PPP user ID, and password */
    NetConfigPPP("username", “password”);
if (NetInit() < 0)
/* process error */


.......



/* shut down all network connections */

NetDeinit();
}
```

**Listing 20.2 Networking API calling sequence**

```
/***************************************************************/
 /* Setup the low level driver/port, which must be set before the
"NetInit()".*/
```

```
          NetConfigModem(UART_2, PPSMSCI, "115200");

// Please make sure hardware flow control is turned on
// in netsci.c

/* Use the profile created */

        NetConfigMachineName("machineABC");

        NetConfigISPAccount(profileArray[profileIndex]->userName,
                profileArray[profileIndex]->password,
                profileArray[profileIndex]->phoneNumber);

/* If the PPP required, setup the PPP user ID, password & the
destination IP address */


        NetConfigPPP(profileArray[profileIndex]->PPPUserName,
                profileArray[profileIndex]->PPPPassword);


}
/* Starts stack, driver */
        NetInit();
```

**Listing 20.3 NTCP File transfer example pseudo-code**

```
/* Example assumes PPSM-GT initialization was done elsewhere */



    /* Client */

    int cs;            /* Socket descriptor for client socket */
    char buff[MAXDAT]; /* Data buffer */
    struct sockaddr_in sock;    /* Socket address structure */
    struct hostent  hostent;    /* Host structure */

    cs = socket(AF_INET, SOCK_STREAM, 0);
    if (cs < 0) /* process error */
    MemMalloc(sizeof(sock));
    sock.sin_family = AF_INET;

    /* Starting with host name, get server IP address */
    gethostbyname_r(SERVER, &hostent, sbuff, sizeof(sbuff),
&stat);
    if (stat < 0) /* process error */
    MemCopy((char *) &sock.sin_addr, (char *)
hostent.h_addr_list[0], 4);

    /* Set the remote port and connect to server */
    sock.sin_port = htons(11001);
    stat = connect(cs, (struct sockaddr *)&sock, sizeof(sock));
    if (stat < 0) /* process error */

    for ( ; ; ) {
        len = fread(ifile, buff, sizeof(buff));
        if (len <= 0)
            break;
        stat = send(cs, buff, sizeof(buff), 0);
        if (stat < 0) /* process error */
    }

    stat = closesocket(cs);
    if (stat < 0) /* process error */


    /* Server */

    int ss;     /* Socket descriptor for server connected socket
*/
    int ls;     /* Socket descriptor for server listening socket
*/
    char buff[MAXDAT];          /* Data buffer */
    struct sockaddr_in sock;    /* Socket address structure */

    ls = socket(AF_INET, SOCK_STREAM, 0);
    if (ls < 0) /* process error */

    /* Bind server socket to port 11001; accept connection from
any remote host */
    MemMalloc(sizeof(sock));
    sock.sin_family = AF_INET;
    sock.sin_addr.s_addr = htonl(INADDR_ANY);
    sock.sin_port = htons(11001);
    stat = bind(ls, (struct sockaddr *)&sock, sizeof(sock));
    if (stat < 0) /* process error */

    /* Listen with backlog of 10; provides for 9 queued requests
*/
    stat = listen(ls, 10);
    if (stat < 0) /* process error */

    /* Accept client connection to new connected socket ss */
    ss = accept(ls, (struct sockaddr *) &sock, &clilen);
    if (ss < 0) /* process error */

    for ( ; ; ) {
        len = recv(ss, buff, sizeof(buff), 0);
        if (len < 0) /* process error */
        if (len == 0) break;
        stat = fwrite(ofile, buff, len);
        if (stat < 0) /* process error */
    }

    /*
    ** Close both connected socket and listening socket. A server
loop
    ** would leave the listening socket open and close only the
connected
    ** socket.
    */
    stat = closesocket(ss);
    stat = closesocket(ls);
```

Section 6

# Graphics & Input Handling Services

This section describes several PPSM-GT features developer can use the PPSM-GT standard services as mentioned in this section to develop the user interface. The PPSM-GT standard graphics and input services provides basic features such as displaying of images and text, drawing of lines and shapes, handling of input pads and software keyboards. The chapters comprising this section are described below:

- Chapter 21, "Graphic Manipulation Services"—explains how PPSM-GT handles graphics and how to program with the graphics API.
- Chapter 22, "Text Management Services"—explains how PPSM-GT handles text and how to program using the text services.
- Chapter 23, "Software Keyboard services"—explains the software keyboard supported by PPSM-GT and how to use it.
- Chapter 24, "Pen Input Handling Services"—explains how PPSM-GT handle the pen input and how to program it.
- Chapter 25, "Handwriting Recognition Input Handling Services"—explains how PPSM-GT handle the 3rd party hand writing recognition engine.

# 21 Graphic Manipulation Services

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

The Graphics services can be grouped under the following main functions:

- Getting and setting of display parameters
  - set grey levels and style
  - set dot width in pixels for drawing dot, line, rectangle, circle, ellipse, arc and vector
  - get information about the LCD display screen.
- Drawing lines and shapes
  - drawing dot, line, rectangle, circle, ellipse, arc and vector
  - draw vector by connecting consecutive points in a list
- Displaying and manipulating of bitmap images
  - swap bitmap images
  - control and restore bitmap images
- Control hardware cursor
  - hardware cursor in inverse and / or blinking mode

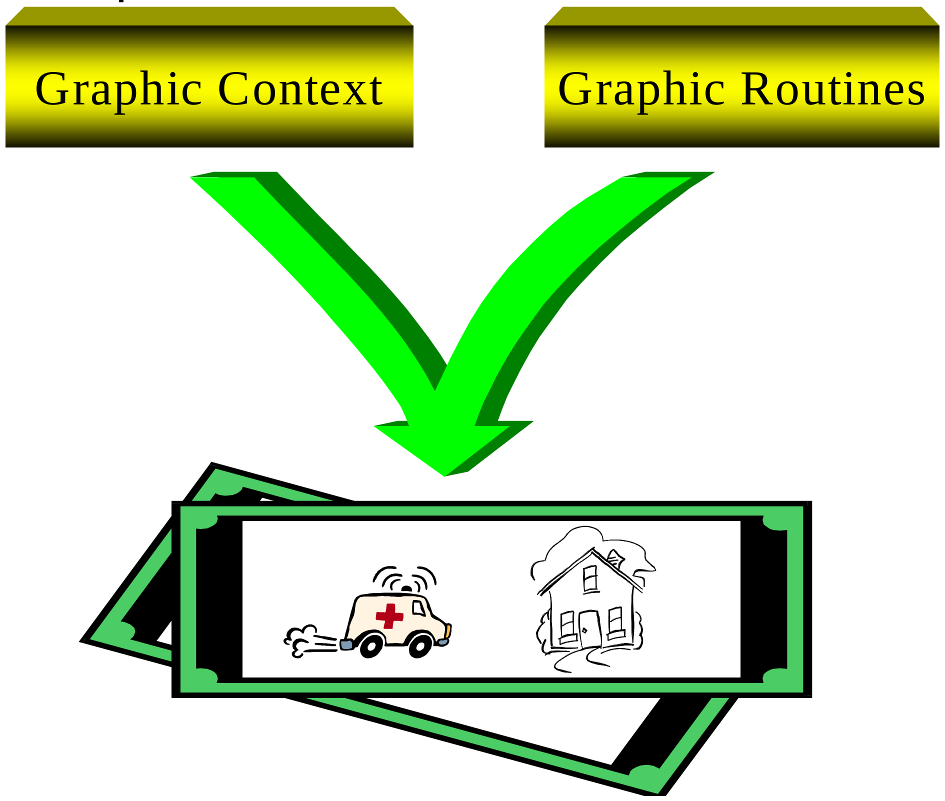
This chapter is organized into the following main sections:

- Graphic Manpulation Services Fundamentals
- Programming using Graphic Manipulation Services

- Summary
- Code Examples

## Graphic Manpulation Services Fundamentals

**Figure 21.1 Graphic Environment**



In PPSM-GT graphics environment is made up of the graphic context and graphic routines. Figure 21.1 - Graphic Environment illustrates the relationship between the graphic context and graphic tools. In the graphic environment, graphic context works together with graphic routines to generate graphics. Graphic contexts are created by application services API AppCreateGC( ) and used to store the drawing properties. Graphic routines on the other hand are used by tasks to do drawing and display based on the drawing properties.

In the other words, to display any graphics using PPSM-GT graphic manipulation services, the graphic context provides the drawing properties and the location to stored the graphic; panning screen etc, and graphic routines do the drawing.

The concept of Graphic Context, GC is important for implementing PPSM-GT graphic routine. Details of GC are covered in the Chapter 11, “System Application Services.”.

## Display Screen Format

**Figure 21.2 Generic Screen Format**

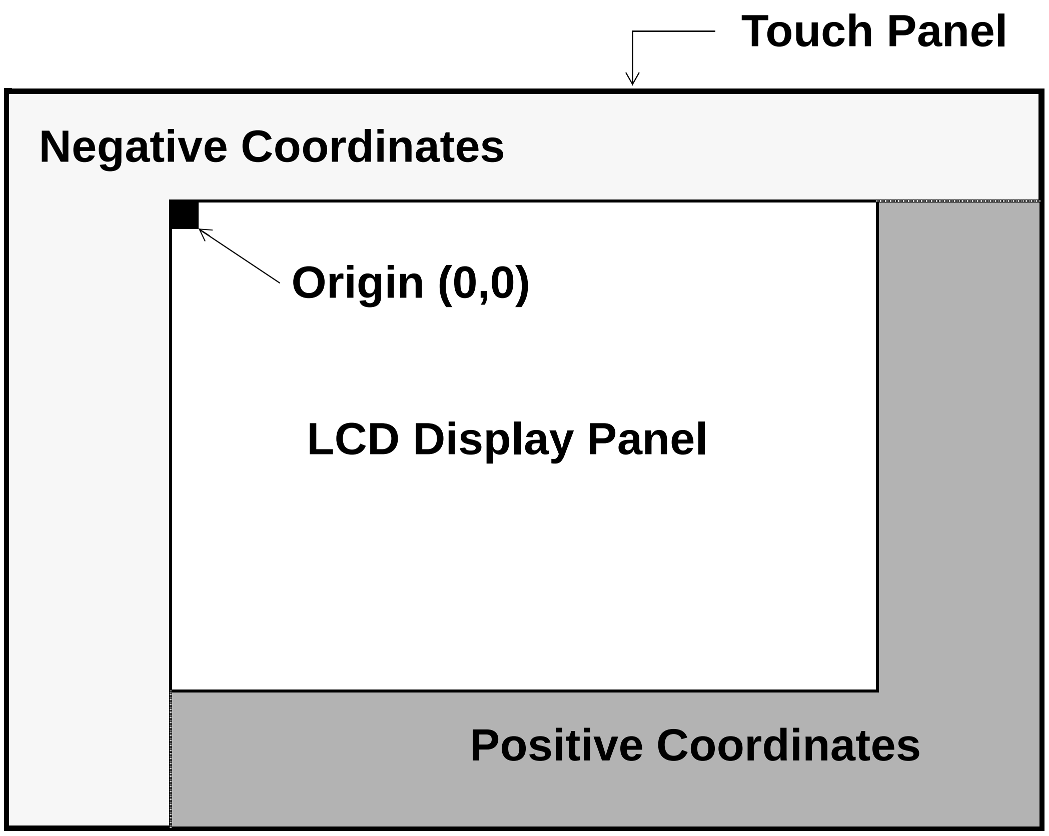


*Figure 21.2* shows the general screen architecture for the PPSM-GT system. There are three major areas:

- LCD Display Screen

- Touch Screen Panel
- Panning Display Screen

## 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, 2 bits per pixel output, or 4 bits per pixel, giving 2 , 4 or 16 grey levels respectively.

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.

## Touch Screen Panel

The touch screen panel is a input device for pen and touch screen inputs. It get it's input coordinates from the LCD display, and need to be calibrated with the LCD display before used.

The touch panel can be larger than the LCD Display Screen. However, the PPSM-GT 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.

## 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 tasks can write to this area, data will not be displayed on the screen unless this area is being mapped to the LCD Display Screen through the panning screen display setting.

Pen Input areas on the panning screen will receive pen input data only when they overlap with the touch panel.

Panning Screen has a similar coordinate system as the LCD Display Screen with different origin.

Panning screens are independent entities and are created when setting up the graphic environment. Refer to Chapter 11, Graphic Manipulation Services for more information on Panning screen.

## Display Coordinates.

The display coordinates in PPSM-GT take reference may the panning screen. That is when displaying any graphic using the graphic services, the top left corner of the panning screen is (0, 0) is taking as the reference. The top left corner of the LCD display screen is therefore an offset from the panning screen origin if it is not at the panning screen origin.

**NOTE** All co-ordinates given to the graphics routines are referred to the panning screen origin. There is no negative co-ordinate for panning screen.

## 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.

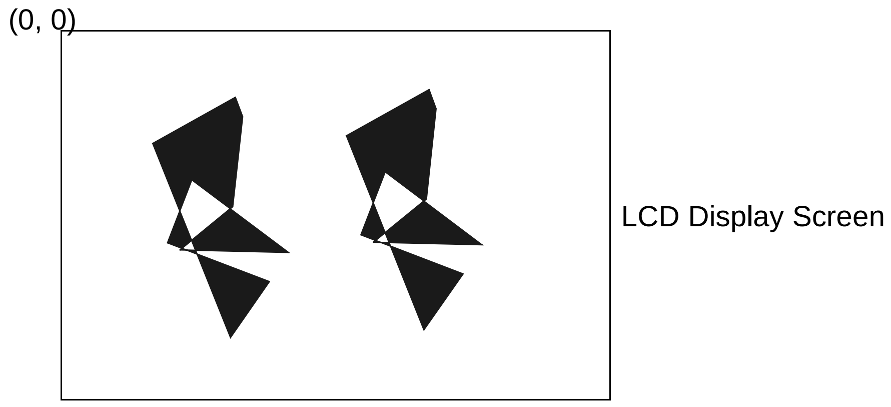
## Screen Initialization

The PPSM-GT returns the LCD coordinates when the touch screen panel is touch, it is therefore important that the LCD screen and the touch screen panel are calibrated to ensure that the correct coordinates are obtained when using the input panel. A screen initialization procedure is required when PPSM-GT is first activated, and can be implemented with PenCalibrate( ). PenCalibrate() is a API in Pen Input Handling Services. Refer to Chapter 24, for more detail on PenCalibrate() API.

## Displaying Color

PPSM-GT supports 3 types of color display; 1 bit, 2 bits, and 4 bits per pixel graphics. Each type operate on it's own pixel library and cannot be mixed. The bitmap formation for each type are different and using the wrong bitmap representation will result in mirror image problem etc. Figure , shows the mirror effects of displaying a 1 bit-per-pixels graphic on a 2 bits per pixel setting.

**Figure 21.3 Effects of display error with wrong pixels**

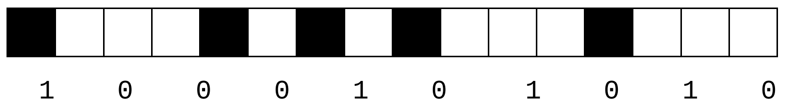


Two similar images will be seen horizontally

### 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:



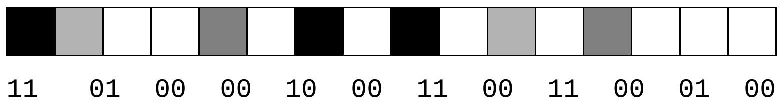
■ BLACK (1) □ WHITE (0)

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

## 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:



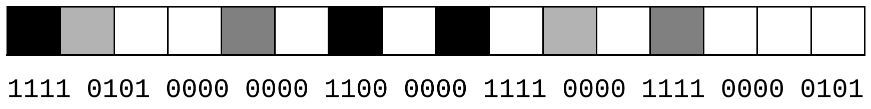
| | | | |
|---|---|---|---|
| ■ | BLACK (11) | ▓ | DARK GREY (10) |
| ░ | LIGHT GREY (01) | □ | WHITE (00) |

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

## 4 bits-per-pixel Graphics

The graphics routine will handle drawing of 16 grey levels: where Grey0 = white and Grey15 =BLACK.

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



| | | | |
|---|---|---|---|
| ■ | BLACK (1111) | ▓ | GREY12 (1100) |
| ░ | GREY5 (0101) | □ | WHITE (0000) |

## Displaying Style

Display style is another properity that could controls the appearance of the output graphics. In PPSM-GT, there are a total of 5 output style that could use to control the graphic outlook. The style mechanism performs a logical operation with the bacground image to achieve the display effects. Table 21.1 shows the 5 style supported and their operation.

Table 21.2 to Table 21.6 provided the logic table based on a 2 bit per pixel graphic representation. It shows 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.

**Table 21.1 Display Style and it's Description**

| Display Style | Description |
|---|---|
| AND_STYLE | Perform AND logical operation with back ground image |
| OR_STYLE | Perform OR logical operation with back ground image |
| EXOR_STYLE | Perform EXOR logical operation with back ground image |
| REPLACE_STYLE | Replace the exiting image with the new image |
| INVERT_STYLE | Perform INVERT logical operation with back ground image |

**Table 21.2 AND_STYLE R = X AND Y after AND_STYLE OPERATION**

| 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 |

**Table 21.3 OR_STYLE R = X OR Y after OR_STYLE OPERATION**

| X | Y | R |
|---|---|---|
| 00 | 00 | 00 |
| 01 | 00 | 01 |
| 10 | 00 | 10 |
| 11 | 00 | 11 |
| 00 | 01 | 01 |
| 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 |

**Table 21.4 EXOR_STYLE R = X EXOR Y after EXOR_STYLE OPERATION**

| 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 |

**Table 21.5 REPLACE_STYLE R = Y after REPLACE_STYLE OPERATION**

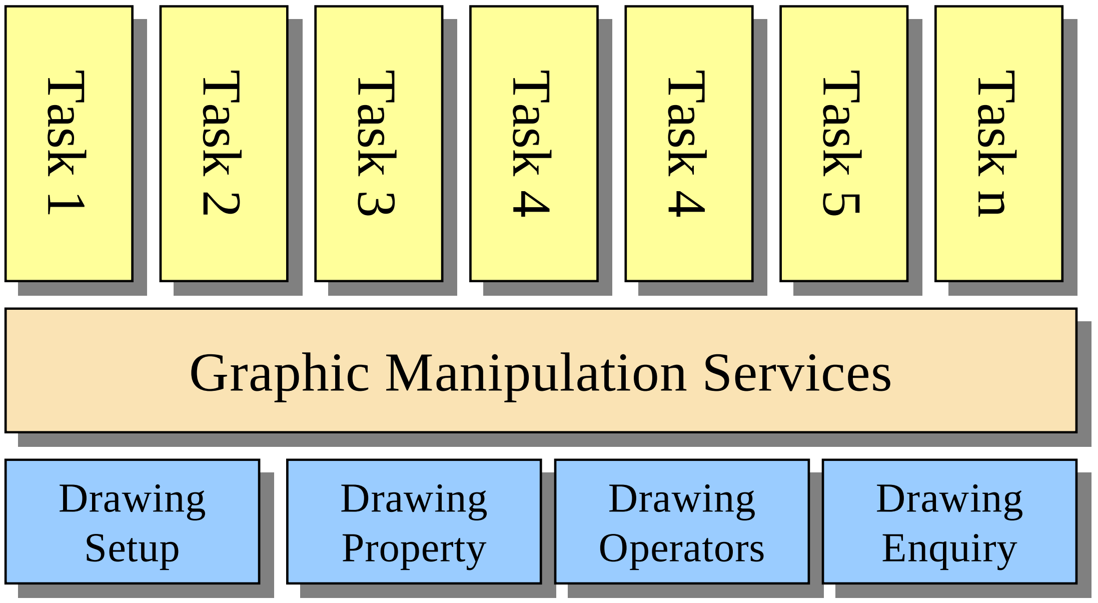
| X | Y | R |
|---|---|---|
| 00 | 00 | 00 |
| 01 | 00 | 00 |
| 10 | 00 | 00 |
| 11 | 00 | 00 |
| 00 | 01 | 01 |
| 01 | 01 | 01 |
| 10 | 01 | 01 |
| 11 | 01 | 01 |
| 00 | 10 | 10 |
| 01 | 10 | 10 |
| 10 | 10 | 10 |
| 11 | 10 | 10 |
| 00 | 11 | 11 |
| 01 | 11 | 11 |
| 10 | 11 | 11 |
| 11 | 11 | 11 |

**Table 21.6 INVERT_STYLE R = NOT X after INVERT_STYLE OPERATION**

| X | R |
|---|---|
| 00 | 11 |
| 01 | 10 |
| 10 | 01 |
| 11 | 00 |

# Programming using Graphic Manipulation Services

**Figure 21.4 Graphic manipulation services Block Dragram**



The graphic manipulation services can be divided into 4 main groups as highlighted in Graphic manipulation services Block Dragram. Tasks interface with the graphic manipulation services through the APIs. The 4 groups of APIs are as followed :

- Drawing Setup
- Drawing Property
- Drawing Operators
- Drawing Enquiry

## Drawing Setup

### Clearing or Filling a Screen

| | |
|---|---|
| STATUS **GpxFillScreen**(COLOR color) | GpxFillScreen() can be used to change the screen color to any of the 16 grey colors in pixels 4 design, 4 colors for 2 pixels design and 2 colors for 1 pixels design. |

### Setting LCD Refresh Rate

| | |
|---|---|
| VOID **GpxSetLCDRefreshRate**(U8 refreshRateSet) | GpxSetRefreshRate() sets the LCD refresh rate in Hz |

### Setting Brightness

| | |
|---|---|
| VOID **GpxSetBrightness**(U8 brightness) | It set the brightness for the LCD screen. |

### Setting Contrast

| | |
|---|---|
| STATUS **GpxSetContrast**(DENSITY level1, DENSITY level2) | It sets the contrast of the LCD |

## Drawing Property

### Setting Color

| | |
|---|---|
| STATUS **GpxSetColor**(COLOR color) | It sets the display color according to the grey scale specified. |

### Setting Style

| | |
|---|---|
| STATUS **GpxSetStyle**(STYLE style) | It sets the style of the graphic context in current task as mentioned in Table 21.1 |

### Setting Dot Width

| | |
|---|---|
| STATUS **GpxSetDotWidth**(U8 width) | After this routine is called, the new dot width will take effect in all subsequent GpxDrawDot(), GpxDrawHorz(), GpxDrawVert(), GpxDrawRec(), GpxDrawLIne(), GpxDrawCircle(), GpxDrawEllipse(), GpxDrawArc(), and GpxDrawVector(). 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 |

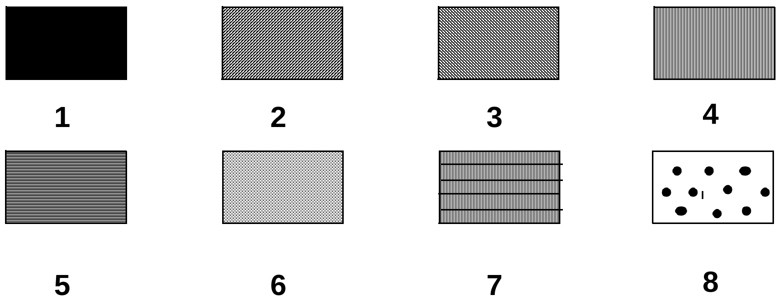
### .Setting Pattern Fill

| | |
|---|---|
| STATUS **GpxSetPatternFill**(U8 mode, COLOR backColor, U8 borderMode, U8 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 GpxSetPatternFill() is called, the settings will be applied to all subsequent GpxDrawRec(), GpxDrawCircle(), GpxDrawEllipse(), and GpxDrawArc(). |

The pattern will be drawn with the specified grey level in the parameter of GpxDrawRec(), GpxDrawCircle(), GpxDrawEllipse(), and GpxDrawArc().

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.

## Hardware Cursor

The following are the steps for setting up the hardware cursor:

- **Step 1:** STATUS **GpxInitCursor**(SCREEN_ID screenId)

  When call, the API will create the cursor (in transparent mode)

- **Step 2:** STATUS **GpxSetCursorSize**(SCREEN_ID screenId, U8 cursorWidth, U8 cursorHeight)

  When call, the API will set the size of the cursor. This routine will set the hardware cursor width and height. The valid range for both width and height is from 1 through 31.

- **Step 3:** STATUS **GpxSetCursorPos**(SCREEN_ID screenId, U16 xPos, U16 yPos)

  When call, the API will set the top left corner of the cursor

- **Step 4**: STATUS **GpxSetCursorStatus**(SCREEN_ID screenId, U8 status)

  When call, the API will turn on the cursor or change its mode to one of the following mode

  – LCD_CURSOR_OFF - Transparent, cursor is disabled

– LCD_CURSOR_ON - Full (black) cursor
– LCD_CURSOR_REVERSED - Reversed video
– LCD_CURSOR_ON_WHITE - Full (white) cursor.

When there is no hardware cursor in current task, the creation of hardware cursor requires to set the cursor characteristic and follows by following the 4 steps above.

When hardware cursor is created and it needs to be turned off, the application should call GpxSetCursorStatus(screenId, LCD_CURSOR_OFF). If the application needs to turn on the cursor once again with the same cursor characteristic. Then calling CursorSetStatus(screenId, LCD_CURSOR_ON) is enough.

When hardware cursor is to be suspended, the application should call GpxCursorSetStatus(screenId, LCD_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.

```
GpxSetCursorStatus(panSc, LCD_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.

### Set Hardware Cursor Blinking Frequency

| | |
|---|---|
| STATUS **GpxSetCursorBlink**(SCREEN_ID screenId, U8 frequency) | This routine will set the hardware cursor blinking frequency to “frequency” number of blinks per 10 seconds. The cursor will only be seen if the cursor is set on by calling GpxSetCursorStatus(). |

### Deleting Hardware Cursor

| | |
|---|---|
| STATUS **GpxDeleteCursor**(SCREEN_ID screenId) | This routine will turn off the hardware cursor permanently and delete the cursor data structure. In order to turn on the hardware cursor again, the application has to follow the 4 steps to setup the characteristics of the cursor mentioned in Hardware Cursor. |

## Drawing Operators

### Drawing a Dot

| | |
|---|---|
| STATUS **GpxDrawDot**(U16 xPos, U16 yPos) | This routine will draw a dot at the specified position (xPos, yPos).<br>If dot width is 1, a pixel will be drawn. Depending on the dot width set by GpxSetDotWidth(), 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).<br>All units specified are in pixels with reference to the panning screen origin. |

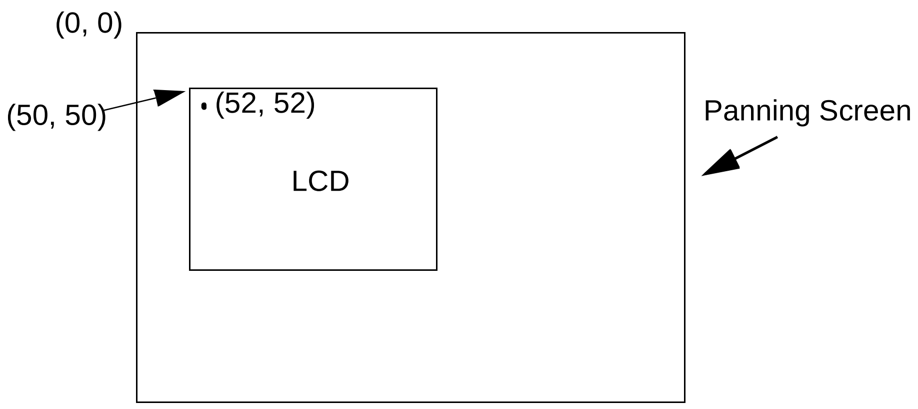


**Figure 21.5 Screen output**

### Drawing a Line

| | |
|---|---|
| STATUS **GpxDrawLine**(U16 xSrc, U16 ySrc, U16 xDest, U16 yDest, U16 dotLine) | This routine will draw a line from (xSrc, ySrc) to (xDest, yDest).<br>All units specified are in pixels with reference to the panning screen origin. |

If the dot width is greater than 1, the specified line will have integer truncated of (dot width - 1)/2 lines above it, and (dot width)/2 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 line is 1, a square dot will be drawn.

## Drawing a Rectangle

| | |
|---|---|
| STATUS **GpxDrawRec**(U16 xSrc, U16 ySrc, U16 xDest, U16 yDest, U16 dotLine) | This routine draws a rectangle with top left corner at (xSrc, ySrc) and bottom corner at (xDest, yDest).<br>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.<br>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.<br>If fill pattern mode is set and border mode is off, the area inside and on the rectangle border will be filled.<br><br>All units specified are in pixels with reference to the panning screen origin. |

## Drawing a Circle

| | |
|---|---|
| STATUS **GpxDrawCircle**(U16 xCenter, U16 yCenter, U16 radius) | This routine will draw a circle centering at (xCenter, yCenter) with the specified radius.<br>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.<br>If both fill pattern mode and border mode are set, those area inside the circle which is not covered by border will be filled.<br>If fill pattern mode is set and border mode is off, the area inside and on the circle border will be filled.<br><br>All units specified are in pixels with reference to the panning screen origin. |

### Drawing an Ellipse

| | |
|---|---|
| STATUS **GpxDrawEllipse**(U16 xCenter, U16 yCenter, U16 xLength, U16 yLength) | This routine will draw a ellipse centering at (xCenter, yCenter) with the specified size.<br>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.<br>If both fill pattern mode and border mode are set, those area inside ellipse which is not covered by the border will be filled.<br>If fill pattern mode is set and border mode is off, the area inside and on the ellipse border will be filled.<br>All units specified are in pixels with reference to the panning screen origin. |

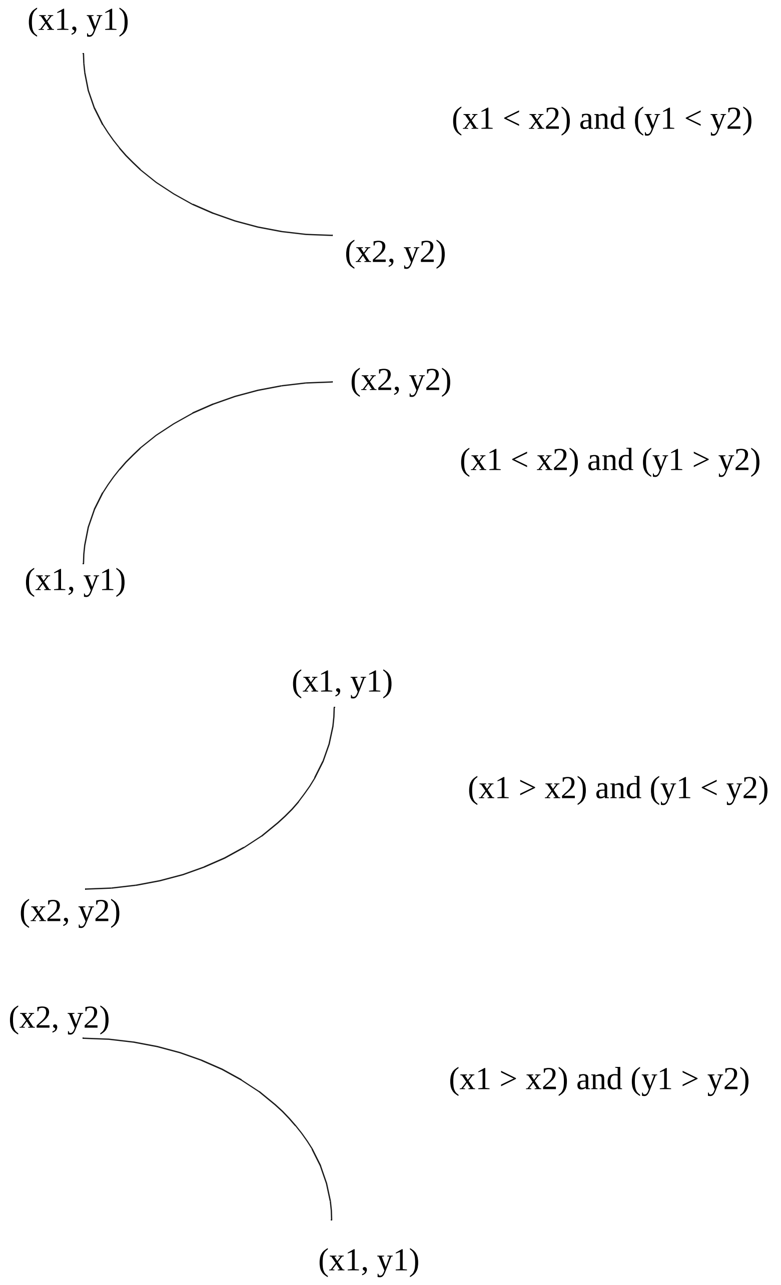
### Drawing a Vector

| | |
|---|---|
| STATUS **GpxDrawVector**(U16 numberOfPoints, P_POINT pPoints, U8 mode) | It connects all points in the vector according to the sequence.<br>All units specified are in pixels with reference to the panning screen origin. |

### Drawing an Arc

| | |
|---|---|
| STATUS **GpxDrawArc**(U16 x1, U16 y1, U16 x2, U16 y2) | This routine will draw an arc connecting (x1, y1) and (x2, y2).<br>GpxDrawArc() will draw a quarter of an ellipse centering at (x2, y1). If GpxDrawArc( x1, y1, x2, y2) is called, the following arcs will be drawn according to the values of (x1, y1) and (x2, y2).<br>All units specified are in pixels with reference to the panning screen origin. |

**Figure 21.6 Cases of GpxDrawArc**



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.

### Putting a Rectangular Area on Panning Screen

| | |
|---|---|
| STATUS **GpxPutRec**(P_U8 pBitmap, U16 xSrc, U16 ySrc, U16 xDest, U16 yDest) | This routine puts an image from memory to panning screen. |

### Special cases of PutRec()

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

#### *LCD Display screen crosses the right boundary of the panning screen*

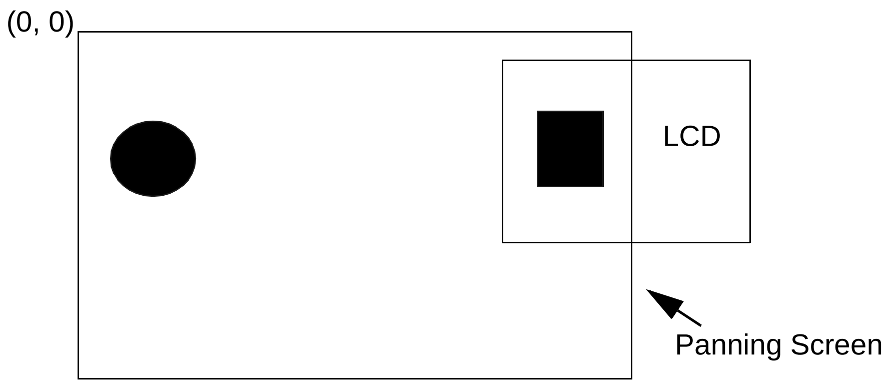


**Figure 21.7 Right Boundary Effect**

The following will be seen on LCD display screen:

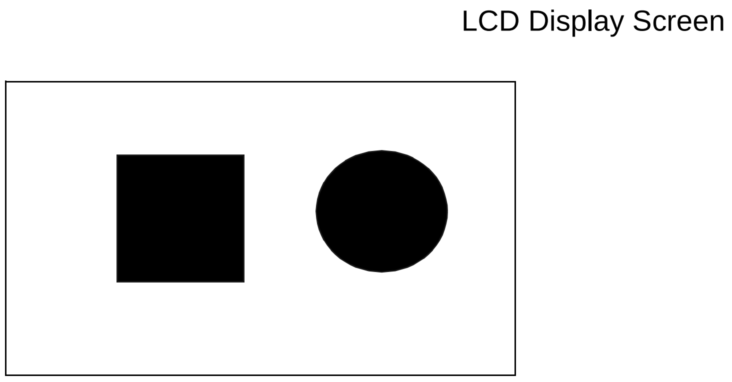


**Figure 21.8 Result of Right Boundary**

### When LCD Display screen crosses the bottom boundary of the panning screen

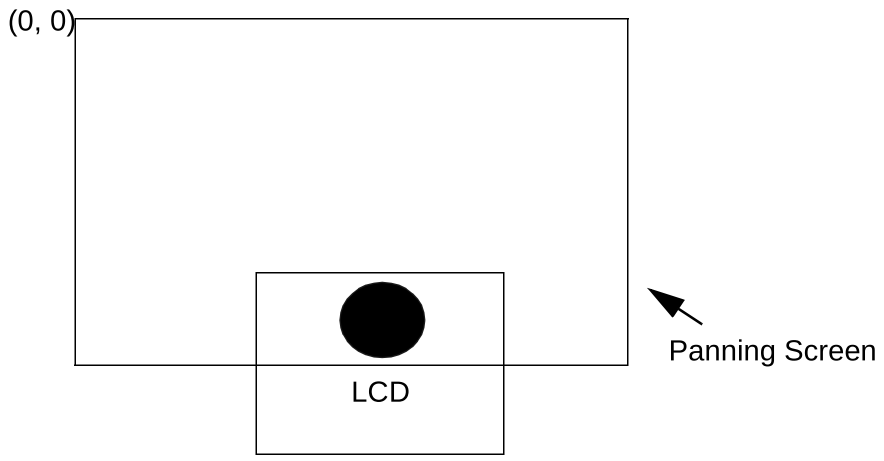


**Figure 21.9 Bottom Boundary Effect**

The following will be seen:

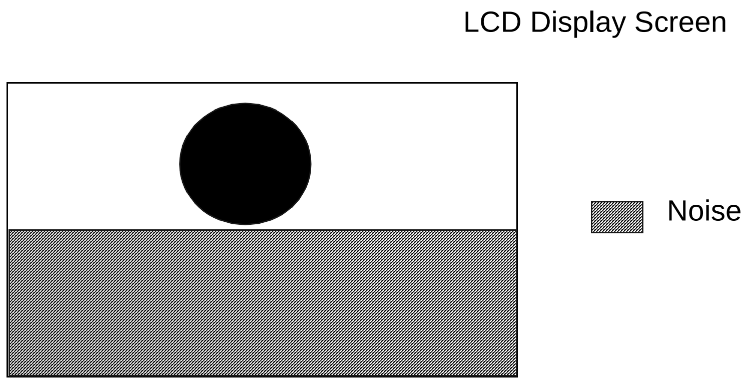


**Figure 21.10 Result of Bottom Boundary**

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.

### Save a Rectangular Area from Panning Screen

| | |
|---|---|
| STATUS **GpxSaveRec**(P_U8 pBitmap, U16 xSrc, U16 ySrc, U16 xDest, U16 yDest) | This routine saves an image from the panning screen to memory. |

### Exchange a Rectangular area with memory

| | |
|---|---|
| STATUS **GpxExchangeRec**(P_U8 pBitmap, U16 xSrc, U16 ySrc, U16 xDest, U16 yDest) | This routine exchanges images between the panning screen and memory. |

### Fill a Rectangular Area

| | |
|---|---|
| STATUS **GpxFillRec**(U16 xSrc, U16 ySrc, U16 xDest, U16 yDest) | This routine fills an rectangular area with the specified grey level. |

### Inverse a Rectangular Area

| | |
|---|---|
| STATUS **GpxInvRec**(U16 xSrc, U16 ySrc, U16 xDest, U16 yDest) | This routine will inverse the grey level of the rectangular area with top left corner at (xSrc, ySrc) and bottom right corner at (xDest, yDest). |

### Display Other Region of Panning Screen or Display Move

| | |
|---|---|
| STATUS **GpxChangeDisplay**(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. The effect is like shift the LCD screen relative to the xPos and yPos specified. |

### Setting Display Origin

| | |
|---|---|
| STATUS **GpxSetDisplayOrigin**(SCREEN_ID screenId, U16 xPos, U16 yPos) | Set the LCD display to new location as specified by XPos & yPos |

The purpose of GpxSet DisplayOrgin( ) is to provide a mean for designer to change the LCD display from one area of the panning screen to another. This is a two step approach to display other region of the same panning screen or other panning screen. These two routines together will achieve the same effect as GpxChangeDisplay().

The advantages of using a step approach instead of a single step approach is that:

it allow the control of the display change. In the two step approach, display change could be control at application switch.

Choice of panning screen based on the screenId.

GpxSetDisplayOrigin() sets the top left corner coordinate (xPos, yPos) of the display origin in reference to the specified panning screen

GpxSetDisplayOrigin()  move the LCD display origin to (x, y) so that different regions of the panning screen and will be displayed instantaneously when the Appswitch() or AppBindPanInfo() is called.

Note that display will not show the panning screen area until Appswitch() or AppBindPanInfo() is call.

The two routines must be used together to have effect on the display.

### Putting Char on Image

| | |
|---|---|
| STATUS **GpxPutChar**(P_U8 pChar, U16 xPos, U16 yPos, U16 font, U16 width, U16 height) | This routine will put the character bitmap onto the image. It does not save the background image. To preserve the background, GpxSaveRec() has to be used. |

## Drawing Enquiry

### Getting LCD Brightness Setting

| | |
|---|---|
| U8 **GpxGetBrightness**(VOID) | This routine returns an 8 bit values from PWM Contrast Control Register. |

### Getting LCD Contrast Setting

| | |
|---|---|
| STATUS **GpxGetContrast**(P_DENSITY pLevel1, P_DENSITY pLevel2) | This routine returns the LCD contrast setting. |

### Getting Current Drawing Property Setting

| | |
|---|---|
| STATUS **GpxGetColor**(P_COLOR pColor) | Get the current color setting. |
| STATUS **GpxGetStyle**(P_STYLE pStyle) | Get the current style |
| STATUS **GpxGetDotWidth**(P_U8 pWidth) | Get the dot width |
| STATUS **GpxGetPatternFill**(P_U8 pMode, P_COLOR pBackColor, P_U8 pBorderMode, P_U8 pFillSpace) | Get the pattern fill. |

## Getting LCD Display Screen Info

### Get LCD Display Screen Width

| | |
|---|---|
| U16 **GpxGetDisplayWidth**(void) | GpxGetDisplayWidth( ) returns to the caller the physical width, in terms of pixels, of the LCD display panel being used.<br>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. |

### Get LCD Display Screen Height

| | |
|---|---|
| U16 **GpxGetDisplayHeight**(void) | GpxGetDisplayHeight( ) returns to the caller the physical height, in terms of pixels, of the LCD display panel being used.<br>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 |

.

### Getting the LCD display Origin

| | |
|---|---|
| STATUS **GpxGetDisplayOrigin**(SCREEN_ID screenId, P_U16 pXPos, P_U16 pYPos) | The routine returns the top left corner coordinate (*pXPos, *pYPos) of the display origin of the specified panning screen. |

### Getting the LCD Refresh Rate

| | |
|---|---|
| U8 **GpxGetLCDRefreshRate**(void) | This routine returns the LCD refresh rate in Hz. |

## Getting Hardware Cursor Info

### Getting Hardware Cursor Position

| | |
|---|---|
| STATUS **GpxGetCursorPos**(SCREEN_ID screenId, P_U16 pXPos, P_U16 pYPos) | This routine returns the top left coordinate (*pXPos, *pYPos) of the hardware cursor of the specified panning screen |

### Getting Hardware Cursor Status

| | |
|---|---|
| STATUS **GpxGetCursorStatus**(SCREEN_ID screenId, P_U8 pStatus) | This routine will return the current hardware cursor status. The status will be one of the following states: LCD_CURSOR_OFF, LCD_CURSOR_ON LCD_CURSOR_REVERSED, or LCD_CURSOR_ON_WHITE. |

## Summary

The graphic services of PPSM-GT provide a mean for developer to draw simple graphics and display bitmaps. Developer could use the graphics services to support the user interface design.

## Code Examples

### Listing 21.1 Fill the whole screen with BLACK

```
STATUS ret;
```

```
/* fill the whole panning screen with black */
ret = GpxFillScreen(BLACK);
```

**Listing 21.2 Initialize screen through PenCalibrate( )**

```
main()
{
     /* Initialize PPSM-GT with pen calibration */
     PenCalibrate(TRUE);
```

**Listing 21.3 Fill the whole screen with BLACK**

```
STATUS ret;

/* fill the whole panning screen with black */
ret = GpxFillScreen(BLACK);
```

**Listing 21.4 Setting the LCD refresh rate at 50 Hz**

```
GpxSetLCDRefreshRate(50);
```

**Listing 21.5 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.
.
.
.
      GpxSetDotWidth(1);
      GpxSetPatternFill(2, WHITE, 0, 3);
```

**Listing 21.6 When there is no hardware cursor in current task**

```
SCREEN_ID   panSc;
/* create the hardware cursor data structure
*/
GpxInitCursor(&panSc);

/* set hardware cursor width to 15 and height to 15
pixels */
GpxSetCursorSize(panSc, 15, 15);

/* set hardware cursor position at (150, 150) */
GpxSetCursorPos(panSc, 150, 158);


/* turn on the hardware cursor in full density
mode */
GpxSetCursorStatus(panSc, LCD_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.

**Listing 21.7 When the hardware cursor needs to be changed to other position**

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

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

**Listing 21.8 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
*/
GpxSetCursorStatus(panSc, LCD_CURSOR_REVERSED);
```

**Listing 21.9 Draw a black dot**

```
STATUS ret;

/* draw a dot at (52, 52) */
     GpxSetColor(BLACK);
     GpxSetDotWidth(1);
     GpxSetStyle(REPLACE_STYLE);
ret = GpxDrawDot(52, 52);
```

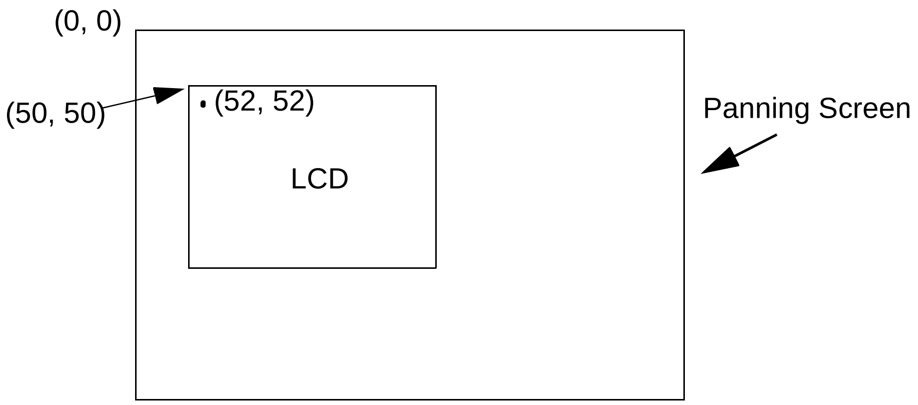


**Figure 21.11 Screen output**

The calling of GpxDrawDot( 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.

**Listing 21.10 Draw a dot with dot width 2**

```
STATUS ret;

/* set dot width to 2 */
GpxSetDotWidth(2);

/* draw a dot at (52, 52) */
ret = GpxDrawDot(52, 52);
```

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

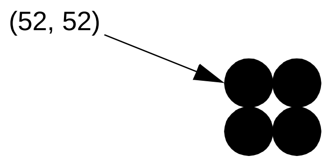


**Figure 21.12 Screen output**

### Listing 21.11 Draw a dot with dot width 3

```
STATUS ret;

/* set dot width to 3 */
GpxSetDotWidth(3);

/* draw a dot at (52, 52 */
ret = GpxDrawDot(52, 52);
```

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

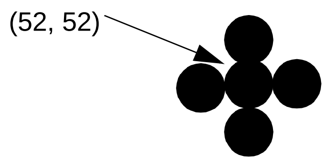


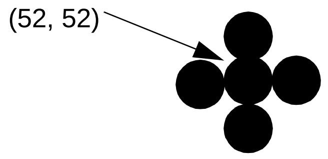
**Figure 21.13 Screen output**

### Listing 21.12 Draw a dot with dot width 4

```
STATUS ret;

/* set dot width to 4 */
SetDotWidth(4);

/* draw a dot at (52, 52) */
ret = GpxDrawDot(52, 52);
```

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

**Figure 21.14 Screen output**

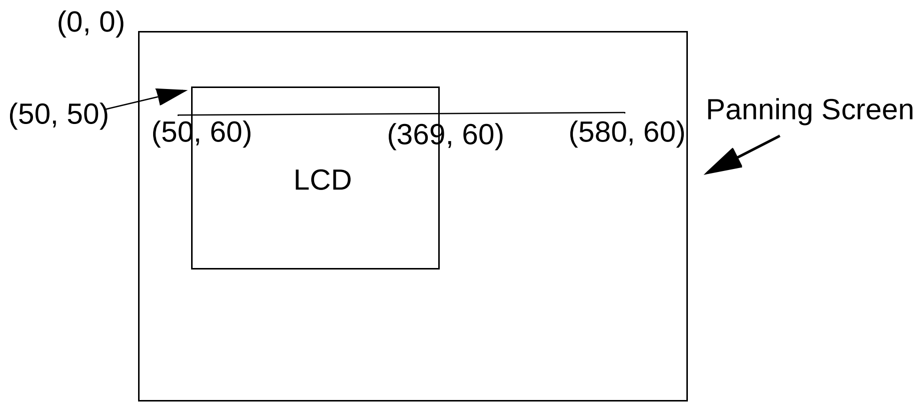


**Listing 21.13 Draw a horizontal black line**

```
STATUS ret;
     GpxSetColor(BLACK);
     GpxSetDotWidth(1);
     GpxSetStyle(REPLACE_STYLE);
/* draw a black horizontal line from (30, 60) with
width 550 */
ret = GpxDrawLine(30, 60, 579, 60,0);
```

In the example, the dot width is 1. The calling of GpxDrawLine( 30, 60, 579, 60, 0) will draw a black horizontal line from (30, 60) to (580, 60) on panning screen. Only the portion of (50, 60) to (580, 60) will be seen on LCD display.

**Figure 21.15 Screen output for drawing a black line**



**Listing 21.14 Draw a thick horizontal line**

```
STATUS ret;

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

if (ret !=SYS_OK)
return ret;

/* draw a black horizontal line from (60, 60) with
width 2 */
ret = GpxDrawLine(60, 60, 61, 60,0);

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

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

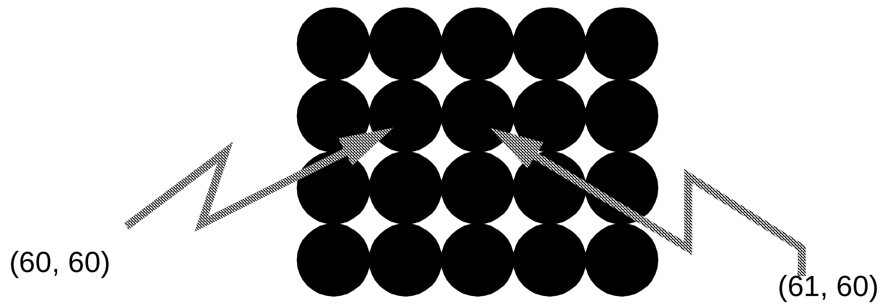


**Figure 21.16 Screen output**

### Listing 21.15 Draw a vertical black line

```
STATUS ret;

/* draw a black vertical line from (60, 60) with
height 360 */
ret = GpxDrawLine(60, 60, 60, 420,0);
```

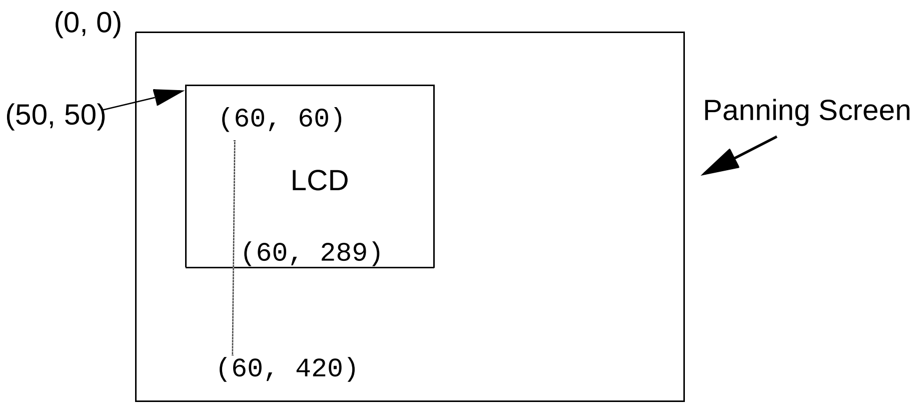


**Figure 21.17 Screen output**

In this example, the dot width is 1. The calling of GpxDrawLine( 60, 60, 60,420) 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.

**Listing 21.16 Draw a thick vertical line**

```
STATUS ret;

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

if (ret !=SYS_OK)
return ret;

/* draw a black thick horizontal line from (10,
10) with height 2 */
ret = GpxDrawLine(10, 10, 10,12,0);

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

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

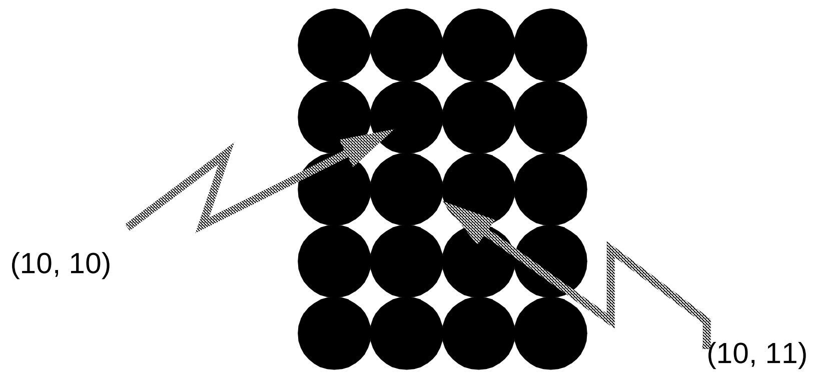


**Figure 21.18 Screen output**

**Listing 21.17 Draw a black line**

```
STATUS ret;
```

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

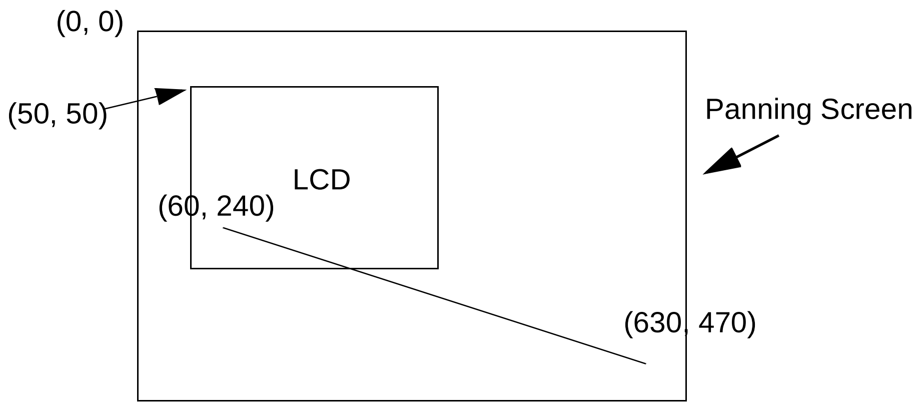


**Figure 21.19 Screen output**

In this example, the dot width is 1. The calling of GpxDrawLine( 60, 240, 630, 470, 0) 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.

**Listing 21.18 Draw a thick line**

```
STATUS ret;

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

if (ret !=SYS_OK)
return ret;

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

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

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

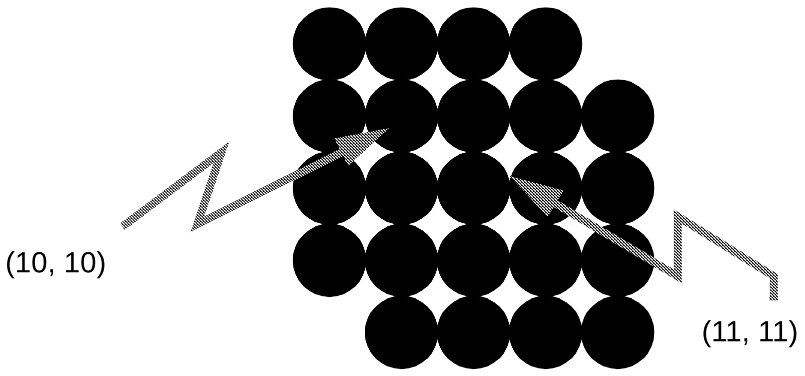


**Figure 21.20 Screen output**

**Listing 21.19 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)
*/
     GpxSetColor(BLACK);
     GpxSetDotWidth(1);
     GpxSetStyle(REPLACE_STYLE);
ret = GpxDrawRec(310, 250, 500, 400, 0);
```

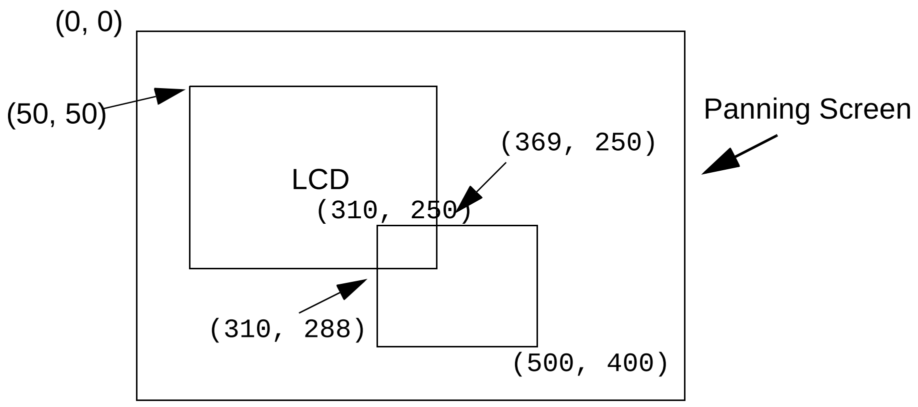


**Figure 21.21 Screen output**

In this example, the dot width is 1. The calling of GpxDrawRec( 310, 250, 500, 400, 0) 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.

**Listing 21.20 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);

if (ret !=SYS_OK) return ret;

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

if (ret !=SYS_OK) return ret;

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

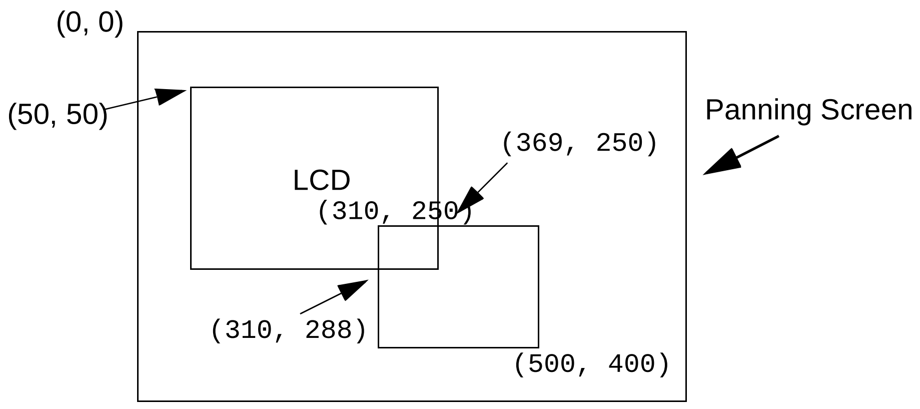


**Figure 21.22 Screen output**

In this example, the dot width is 3 and fill Pattern mode is 1. The calling of GpxDrawRec( 310, 250, 500, 400, 0) 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.

### Listing 21.21 Draw a circle with black outline

```
STATUS ret;

/* draw a black outlined circle with center at
(560, 290) and radius 150 */
     GpxSetColor(BLACK);
     GpxSetDotWidth(1);
     GpxSetStyle(REPLACE_STYLE);

ret = GpxDrawCircle(560, 290, 150);
```

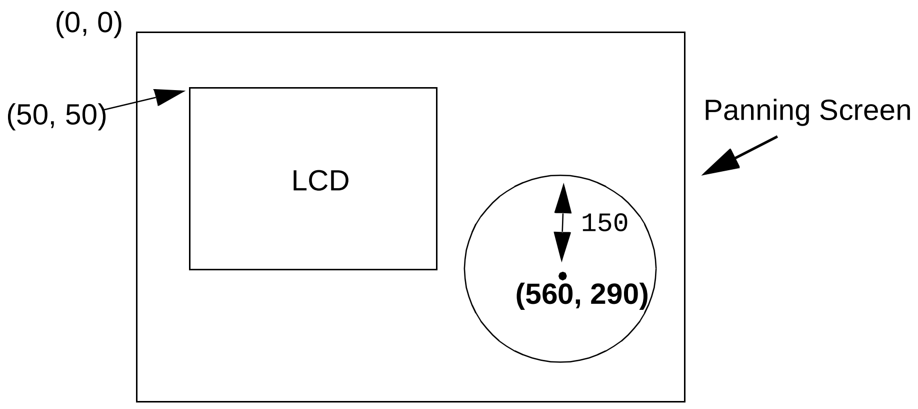


**Figure 21.23 Screen output**

In this example, the dot width is 1. The calling of GpxDrawCircle( 560, 290, 150) 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.

### Listing 21.22 Draw an ellipse with black outline

```
STATUS ret;
```

```
/* draw an ellipse with center at (560, 290),
horizontal length 150 and vertical length 100 */
      GpxSetColor(BLACK);
      GpxSetDotWidth(1);
      GpxSetStyle(REPLACE_STYLE);

ret = GpxDrawEllipse(560, 290, 150, 100);
```

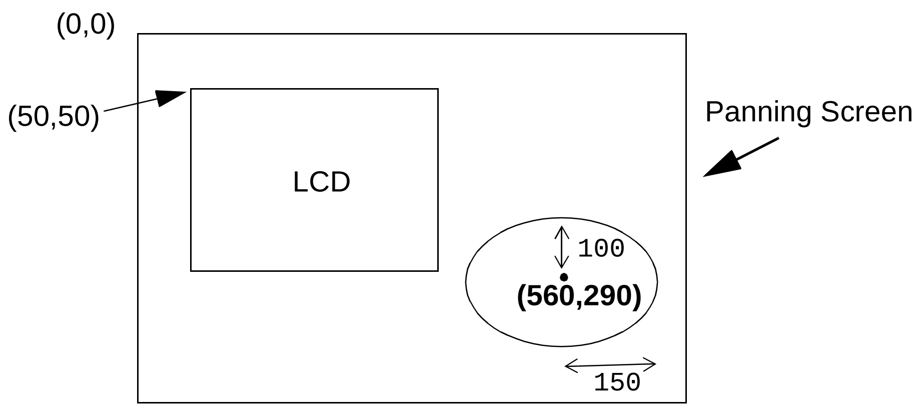


**Figure 21.24 Screen output for *Example***

In this example, the dot width is 1. The calling of GpxDrawEllipse( 560, 290, 150, 100) 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.

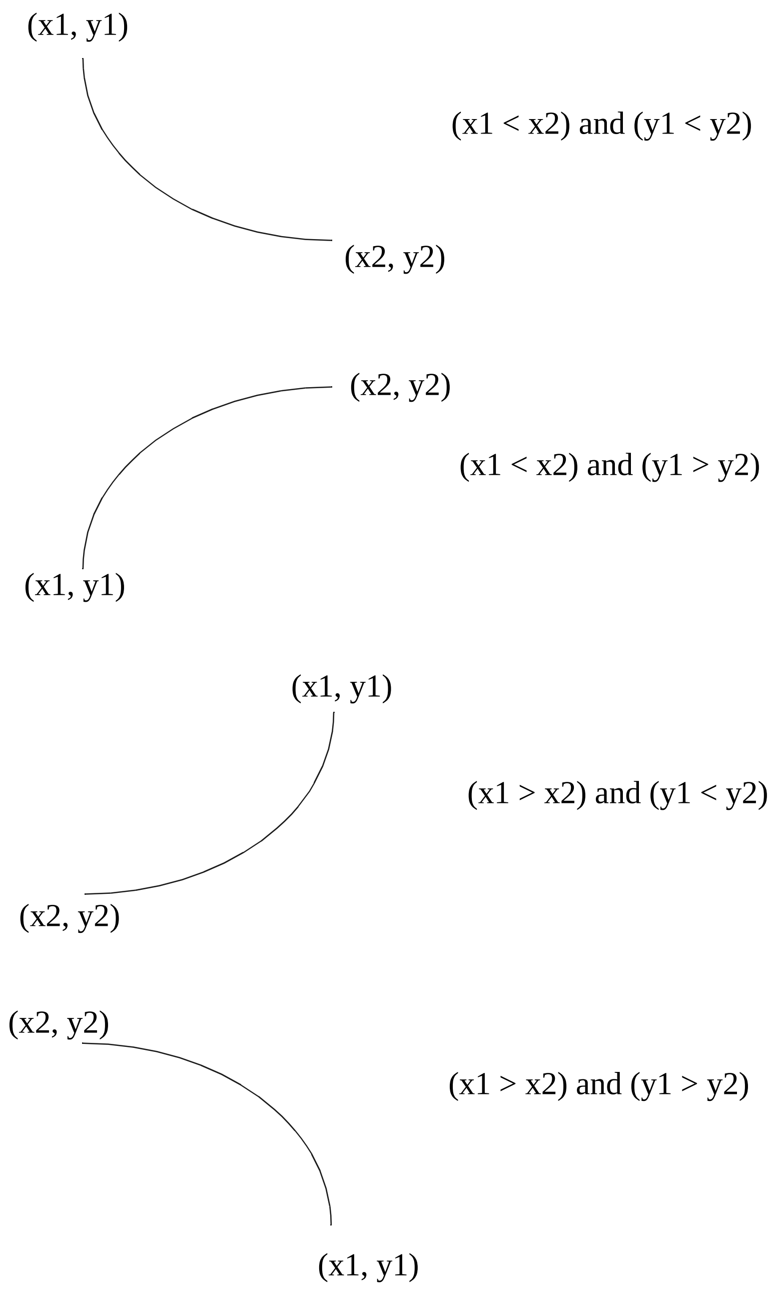


**Figure 21.25 Cases of GpxDrawArc**

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.

**Listing 21.23 Draw a black arc with OR style**

```
STATUS ret;

/* draw an arc from (240, 150) to (100, 100) */
      GpxSetColor(BLACK);
     GpxSetDotWidth(1);
     GpxSetStyle(OR_STYLE);

ret = GpxDrawArc(240, 150, 100, 100);
```

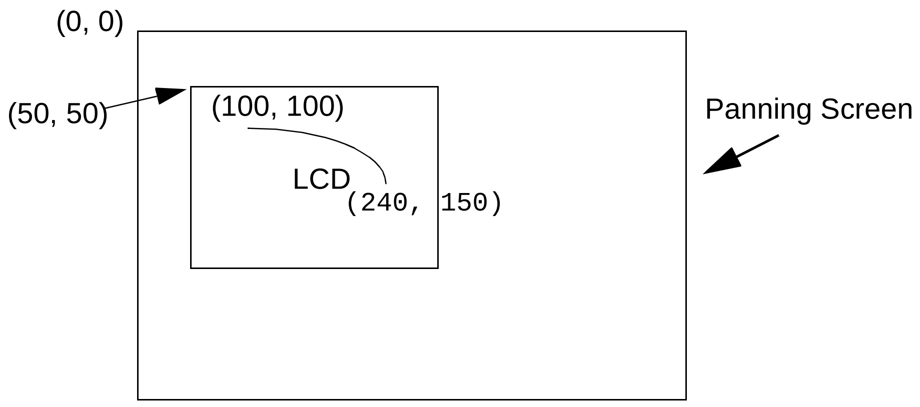


**Figure 21.26 Screen output**

In this example, the dot width is 1. The calling of GpxDrawArc( 100, 100, 50, 50) 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.

### Listing 21.24 Draw a black arc with EXOR style

```
/* draw an arc from (100, 100) to (240, 150) */
     GpxSetColor(BLACK);
     GpxSetDotWidth(1);
     GpxSetStyle(EXOR_STYLE);

ret = GpxDrawArc(100, 100, 240, 150);
```

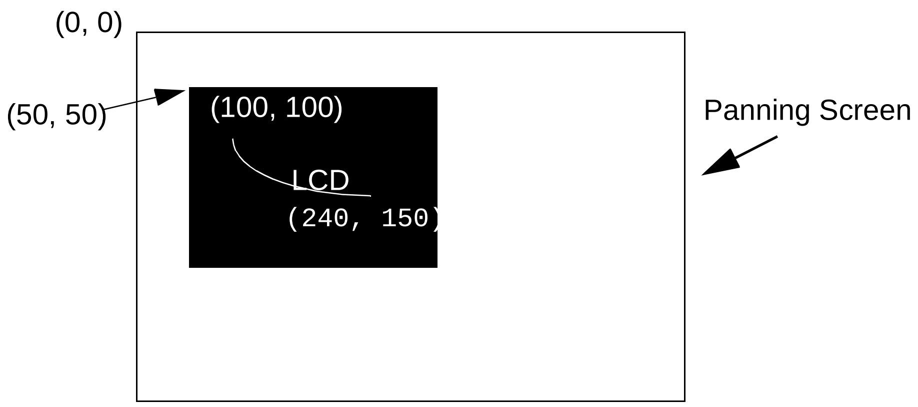


**Figure 21.27 Screen output**

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

### Listing 21.25 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 */
GpxSetStyle(REPLACE_STYLE);


ret = GpxPutRec(&bitmap, 0, 0, 639, 479);
```

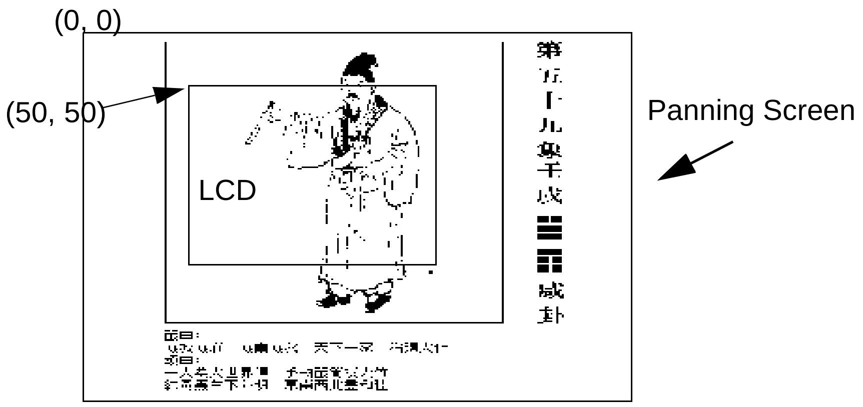


**Figure 21.28 Screen output**

The calling of GpxPutRec(&bitmap, 0, 0, 639, 479) copies the image from the memory area pointed to by *bitmap* onto the panning screen.

**Listing 21.26 Save a bitmap**

```
/* save the portion of image on panning screen
from top left corner at (50, 50, 369, 169), width
320 and height 120 */
ret = GpxSaveRec(&bitmap, 50, 50, 369, 169);
```

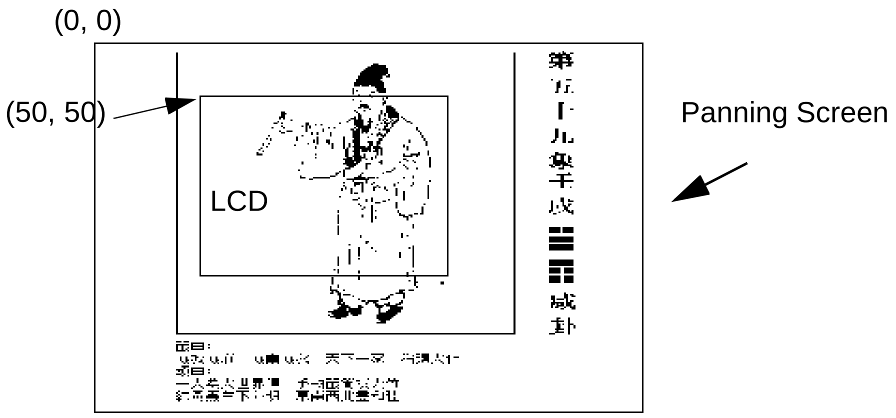


**Figure 16-1 Saving Image 320 x 120**

The calling of GpxSaveRec(&bitmap, 50, 50, 369, 169) will save the top half of LCD display image into memory area pointed to by *bitmap*.

### Listing 21.27 Exchanging 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 newmap */
P_U8   ptrnewmap;

ret = GpxExchangeRec(ptrnewmap, 50, 50, 369, 169);
```

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

### Listing 21.28 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 */
      GpxSetColor(BLACK);
      GpxSetStyle(OR_STYLE);

ret = GpxFillRec(300, 240, 560, 400);
```

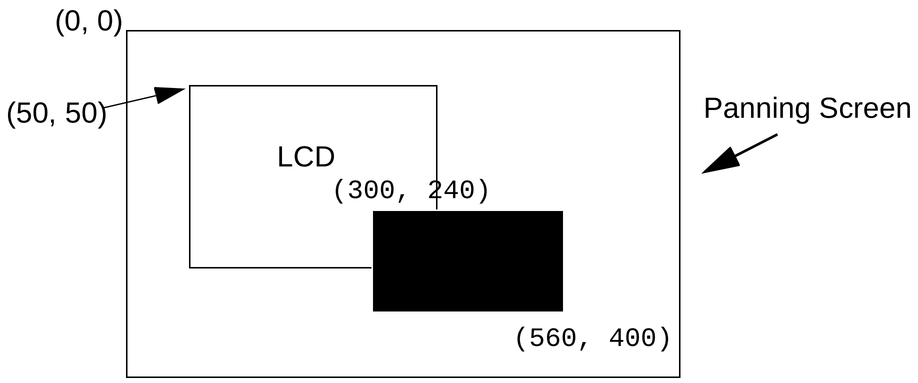


**Figure 21.29 Filling a rectangular region**

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

**Listing 21.29 Inverse a rectangular region**

```
STATUS ret;

/* inverse a rectangular area with top left corner
at (50, 50, 300, 240) */
ret = GpxInvRec(50, 50, 300, 240);
```

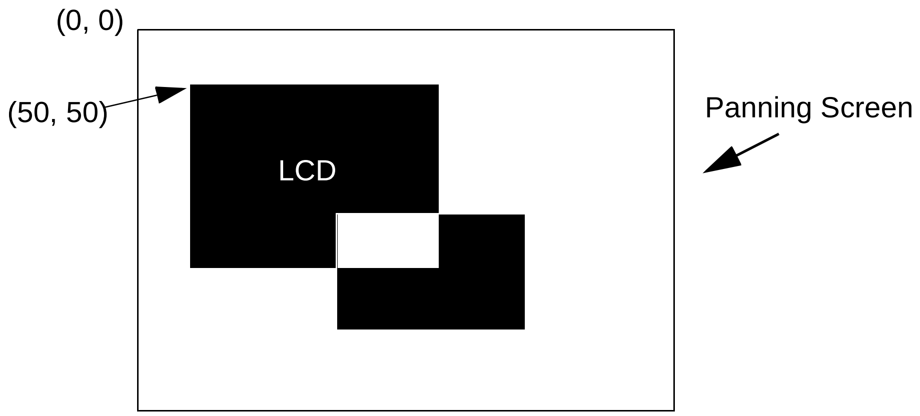


**Figure 21.30 Inverting a Rectangle (50,**

In this example, the LCD display screen is as shown in Figure 21.5 - Screen output before the inversion. After inverting the rectangular region with top left corner at (50, 50, 300, 240) is inverted as shown above.

**Listing 21.30 Display other region of panning screen using GpxChangeDisplay()**

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

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

**Listing 21.31 Display other region of panning screen**

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

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

/* change the hardware register to display the
rectangular on panning screen with top left corner
at (50, 50) */
AppBindPanInfo(appId, newPanId);
```

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

**Listing 21.32 Display the word "rabbit" on the rabbit graphic.**

```
TEXT   Rabbitword[6];
U8     Gpxstorage[1062];
           /* For pixel 4, ((140-39)*(180-159))*4/
8 */
U32    rabbit;
/* Display the word "RABBIT" on the rabbit image
*/

GpxFillScreen(WHITE);
GpxSetStyle(REPLACE_STYLE);
GpxPutRec(&rabbit,0,0,159,239);
GpxSaveRec(Gpxstorage, 38,158,140,180);
Rabbitword[0]='R';
Rabbitword[1]='A';
Rabbitword[2]='B';
Rabbitword[3]='B';
Rabbitword[4]='I';
Rabbitword[5]='T';

for (i=0; i<6; i++)
{
    GpxPutChar(FontGetCharAddr(LARGE_NORMAL_FONT,
Rabbitword[i]), 40+(i*16), 160, LARGE_NORMAL_FONT,
16, 20);
}
```

**Listing 21.33 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 >=
GpxGetDisplayWidth()) || (ySrc < 0) ||
      (yDest >= GpxGetDisplayHeight()) )
          return SYS_error;
```

**Listing 21.34 When the hardware cursor status is needed**

```
U8 status;

/* get the hardware cursor status */
GpxGetCursorStatus(panSc, &status);
```

# 22

# Text Management Services

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-GT to manage the display of text on the panning display screen.

PPSM-GT supports 8-bit and 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-GT 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.

This chapter is organized into the following main sections:

- Text Management Services Fundamentals
- Programming using Text Management Services
- Summary
- Code Examples

## Text Management Services Fundamentals

PPSM-GT text module can handle 8-bit text string, 16-bit text string and 8-bit/16-bit mixed text string. A 8-bit text string means that 8 bits are used to represent a single characters. User should input the character bit length carefully when calling text module API to print text on screen.

## Mixed Font

Text can be displayed with both ASCII and Asian codes together in the same message. PPSM-GT allows text template that carry both font type.

## Text Display Area

Text can be displayed anywhere within the 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 22.1*.

## Text Properties

Text properties describes the layout and appearance of the text to be displayed on the panning display screen. These text properties include the position and size of the text display area, the size, output style and color of the characters and the position of the text soft cursor within the text template.

## Soft cursor

It refers to the position of the next character will be printed on screen. It is different from hard cursor because soft cursor do not display on screen.

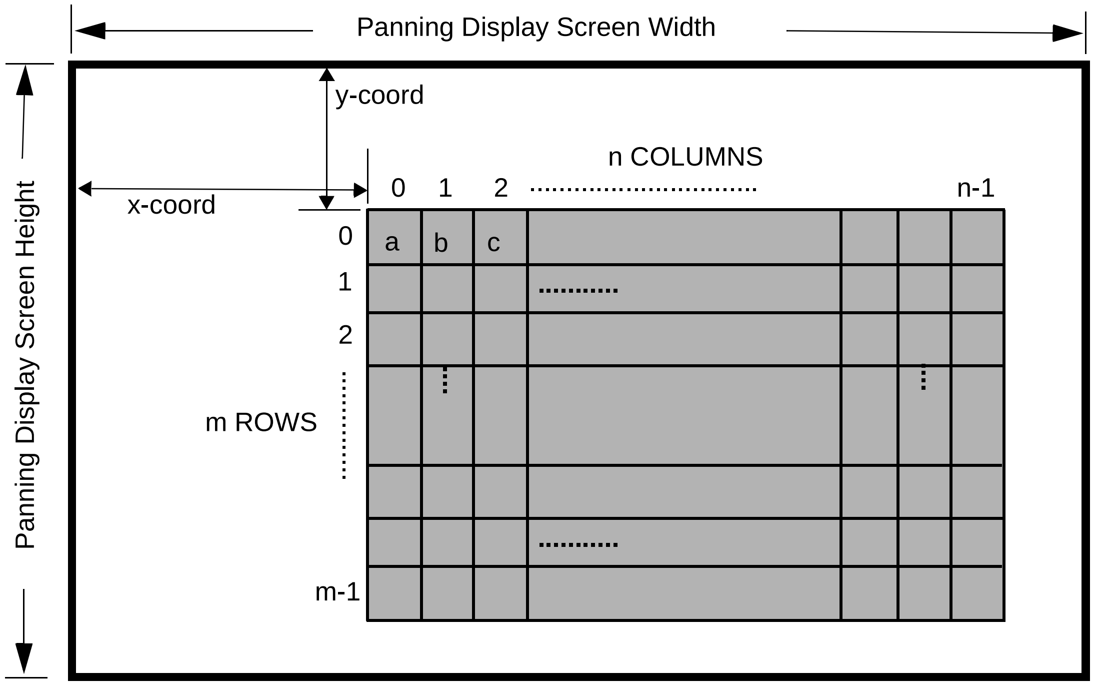
## Sixteen Color Display

Text can be displayed in sixteen different colors. Different sections of the message could also be display with different colors.

## 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.

**Figure 22.1 A Text Display Area on the Panning Display Screen**



In *Figure 22.1*, 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.

# Programming using Text Management Services

## Creating text templates

| | |
|---|---|
| STATUS **TxtCreateTmplt**(P_TMPLT_ID pTemplateId) | A text template needs to be created before any text can be displayed. A unique 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. |

## Default setting of text template

These are the setting once a text template is created by TxtCreateTmplt().

**Table 22.1** **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 |
| Text color | BLACK |
| Text output style | REPLACE_STYLE |

## Deleting text templates

| | |
|---|---|
| STATUS **TxtDeleteTmplt**(TMPLT_ID *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 given to TxtDeleteTmplt() is used to specify which text template to be deleted. |

## Setting Up the Text template

| | |
|---|---|
| STATUS **TxtSetupTmplt**(TMPLT_ID templateId, FONT_TYPE fontType, STYLE outputStyle, COLOR fontColor, U16 xPos, U16 yPos, U16 width, U16 height) | The text template is a rectangular layout that reside within the boundary of the panning screen. The layout is anchored by the xy-coordinate specified by xPos & yPos of the upper left corner, and the width and height of the area in number of characters.<br><br>The size of the text display area in number of pixels will vary according to the size of the selected font type. A 16-bit per character text template have twice the area size as compared to a 8-bit character text template given the same values of width and height.<br><br>The output style is one of the 5 style: (REPLACE_STYLE, AND_STYLE, OR_STYLE, EXOR_STYLE, and INVERSE_STYLE). |

## Setting Template Size

| | |
|---|---|
| STATUS **TxtSetTmpltSize**(TMPLT_ID templateId, U16 width, U16 height) | Set the size of a text template. |

## Setting Template Origin

| | |
|---|---|
| STATUS **TxtSetTmpltOrigin**(TMPLT_ID templateId, U16 xSrc, U16 ySrc) | Put a text template to a specific location on panning screen. |

## Setting Text Output Font Color

| | |
|---|---|
| STATUS **TxtSetFontColor**(TMPLT_ID templateId, COLOR fontColor) | The text color can be set to any of the color that the system support. The new color takes effect on subsequent text mapping on that text template. |

## Setting Text Output Font Style

| | |
|---|---|
| STATUS **TxtSetFontStyle**(TMPLT_ID templateId, STYLE fontStyle) | The output style defines an operation between the text bitmap and the existing image at the same display location. Five output styles are supported. The text bitmap can replace, OR with, AND with, exclusive OR with, or be inverted to the existing image as shown in Table 22.2 |

**Table 22.2** **Supported Output Styles**

| Output Styles | Operation |
|---|---|
| REPLACE_STYLE | Replace |
| OR_STYLE | Or with |
| AND_STYLE | And with |

| Output Styles | Operation |
|---|---|
| EXOR_STYLE | Exclusive-Or with |
| INVERSE_STYLE | Invert and replace |

## Setting Font Type

| | |
|---|---|
| STATUS **TxtSetFontType**(TMPLT_ID templateId, FONT_TYPE fontType) | Four default font types are shipped with PPSM-GT. They are listed in the below table. Small Normal and Small Italic are 8 x 10 pixels English fonts as shown in Table 22.3 |

**Table 22.3** **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 |

**NOTE** Asian fonts are supplied by third parties, but can be integrated with PPSM-GT’s device driver.

## Setting Line Spacing

| | |
|---|---|
| STATUS **TxtSetLineWt**(TMPLT_ID templateId, U16 lineWt) | Set the line width in a text template. It must be larger than the height of the characters to be printed. |

## Text Mapping

| | |
|---|---|
| STATUS **TxtMap**(TMPLT_ID templateId, U8 bitLen, P_TEXT buffer, U16 size) | Mapping functions are provided for applications to display text on the panning display screen area. The display of text are tied to a text template, extra characters are truncated.<br><br>The given text is displayed starting at the current character cursor position of the text display area and with text properties of the text template.<br><br>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. |

## Removing text

| | |
|---|---|
| STATUS **TxtUnmap**(TMPLT_ID *templateId*) | The unmapping of text means clearing the entire text display area in the text template. |

## Text character cursor position

The character cursor position determines where to print the character when next character is mapped to the text template. 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 22.1*, the range of valid cursor positions is zero through (m * n - 1), and the current cursor position is 3 after "abc" is displayed

## Setting the character cursor position

| | |
|---|---|
| STATUS **TxtSetCurPos**(TMPLT_ID 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. |

## Setting the character cursor using X Y coordinates

| | |
|---|---|
| STATUS **TxtSetCurXY**(TMPLT_ID templateId, U16 xPos, U16 yPos) | Setting the character cursor X Y coordinates of the text display area of the specified text template to the given value. Subsequent displaying of text start at this new cursor coordinates. |

## Reading the character cursor position

| | |
|---|---|
| STATUS **TxtReadCurPos**(TMPLT_ID *templateId*, P_U16 *cursor*) | Applications can inquire the current character cursor position of a text template. The returned character cursor position is where text will be displayed next. |

## Printing Text message

| | |
|---|---|
| STATUS **TxtPrintf**(TMPLT_ID templateId, P_U8 pFormatStr, P_VOID argList) | Display the input string according to the format specified in the format string *pFormatStr*. The format string consist of one or several format unit. Each format unit can consist of the following format elements: pre-padding flag, width, precision and type.<br><br>%[Flags][Width][.Precision][Type]<br>– Flags: -, #, 0 or Blank space<br>– Width: Minimum characters must be used to print value<br>– Precision: Minimum no. of decimal places to be printed<br>– Type: d, u, c, C, e, E, f, s, S, x, X<br>– d: Signed decimal<br>– u: Unsigned decimal<br>– c :Character in 8-bit format<br>– C: Character in 16-bit format<br>– e: Exponential Floating-pointer integer<br>– E: Exponential Floating-pointer integer<br>– f: Floating-pointer integer<br>– s: String in 8-bit format<br>– S: String is in 16-bit format<br>– x: Hexadecimal integer<br>– X: Hexadecimal integer<br>– Escape characters<br>- \n: new line<br>- \t: 4 spaces TAB<br>- \": double quote<br>- \': single quote<br>- \\: backslash |

## Summary

Text management services provide a tools and template for handling displaying of text using the PPSM-GT. Text display with PPSM-GT works on a template basis such that a text template is first create before it could be used for displaying the text. This template could then be reused when displaying other texts. The advantage of such a format is that texts are stored in the raw format and appearances changes with the template used.

## Code Examples

**Listing 22.1 Create a text template**

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

    if(TxtCreateTmplt(&tId) != SYS_OK)
      return SYS_ERR;
```

**Listing 22.2 Delete a text template**

```
/* Delete the text when it’s no longer needed */
  if(TxtDeleteTmplt(tId) != SYS_OK)
    return SYS_ERR;
```

**Listing 22.3 Setting text properties**

```
TMPLT_ID tId;/* text template id */
        /* text to be displayed */
TEXTmoto[] = {’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 */
.
.
.
/* create a text template */
```

```
    TextCreate(&tId);

/* subsequent text displayed with text template tId will be small
normal font, OR output style, white in color, located at (10, 20),
length of moto characters wide(64 pixels) and 1 character high(10
pixels), starting at text cursor position zero. */
    TxtSetFontType(tId, SMALL_NORMAL_FONT);
    TxtSetFontStyle(tId, OR_STYLE);
    TxtSetFontColor(tId, WHITE);
    TxtSetTmpltOrigin(tId, 10, 20);
    TxtSetTmpltSize(tId, 64, 10);
    TxtSetCurPos(tId, 0);

/* Map the 16-bit text string to the text template */
    TxtMap(tId, SIXTEEN_BIT, (P_TEXT)moto, 8);

/* delete unused text template */
    TxtDeleteTmplt(tId);
    .
    .
    .
```

**Listing 22.4 Display text on text display area**

```
/*
 *   Prints out message(a row only) on the screen start at (xSrc,
ySrc)
 */
void Typing(U8 font, U8 style, U8 greylev, U16 xSrc, U16 ySrc, U8
bitLen, TEXT str[])
{
    U16 len;
    TMPLT_ID tId;

    /*  create the text template  */
    TxtCreateTmplt(&tId);

    /*  find out the length of the message and print it out  */

    if(bitLen == 8)
    {
        if (len = strlen((P_U8)str))
        {
            TxtSetupTmplt(tId, font, style, greylev, xSrc, ySrc,
len, 1);
            TxtMap(tId, EIGHT_BIT, (P_U8)str, len, 0);
        }
    }
    else
    {
        if (len = Strlen(str))
        {
            TxtSetupTmplt(tId, font, style, greylev, xSrc, ySrc,
len, 1);
            TxtMap(tId, SIXTEEN_BIT, (P_TEXT)str, len, 0);
        }
    }
```

**Listing 22.5 Set character cursor position**

```
TMPLT_ID tId;
.
.
.
/* Clear the text on the display and reset cursor */
    TxtUnmap(tId);
    TxtSetCurPos(tId, 0);
```

**Listing 22.6 Set and read the character cursor position**

```
TMPLT_ID tId;/* text template id */
TEXTmoto[] = {’M’, ’o’, ’t’, ’o’, ’r’, ’o’, ’l’, ’a’, 0};/* text
to be displayed */
U16len;/* # chars to be displayed */
U16curPos;/* cursor position */
.
.
.
/* create a text template */
```

```
TxtCreateTmplt(&tId);

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

/* set up text properties. */
TxtSetupTmplt (tId, SNF, REPLACE_STYLE, BLACK,0,200,19,1);

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

/* display “Motorola” using the modified text properties */
TxtMap(tId, (P_TEXT)moto, len);

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

**Listing 22.7 Example for TxtPrintf()**

```
char str[]="hello";
U16 i = 123;
float f = 4.567;
      TxtPrintf(tId, "abc %s, %d, %4.2f", str, i, f);

/* The output is:
abc Hello, 123, 4.57*/
```

# 23 Software Keyboard services

PPSM supports two types of input methods for applications to receive character input from the user. The chapter provides information on one of the method, the soft keyboard services. The soft keyboard supported by PPSM-GT provides both a default version and customized version.

This chapter is organised into the following sections:

- Soft Keyboard Fundamental
- Programming using Soft Keyboard Services

## Soft Keyboard Fundamental

The soft keyboard function like any standard keyboard except the keyboard is generated through software routine. Data entries are made through the key presses and each key pressed will generate an event message to the target task. The target task is the task that owns the keyboard and all key presses will be communicated to that task.

A default QWERTY soft keyboard with key size of 15x15 pixels can be opened at any position within the panning screen. There are three soft keyboard layouts: one for upper case letters(refer to *Figure 23.1*), one for lower case letters (refer to *Figure 23.2*), and one for numbers and symbols (refer to Figure 23.3).

The 3 layouts are offered together in the default soft keyboard. The up and down arrows keys are for switching between the upper and lower case letters layouts, and the symbol “!@$” switch in the numbers and symbols layout.

**Figure 23.1 Upper Case Soft Keyboard Layout**

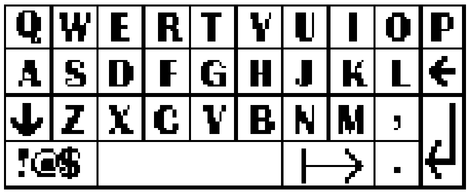


**Figure 23.2 Lower Case Soft Keyboard Layout**

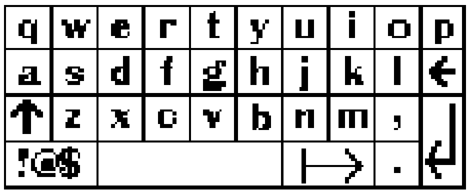
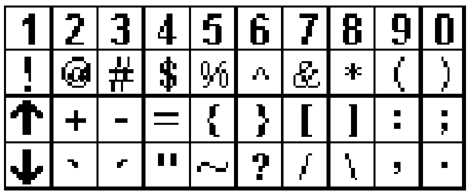


**Figure 23.3 Symbolic Soft Keyboard Layout**



As an alternative, an user may define its own keyboard with required number of column and row of keys and size of each key in number of pixels. User can define the return code of each key and the bitmap of the soft keyboard.

# Programming using Soft Keyboard Services

## Opening Soft KeyBoard

PPSM GT provides 3 ways to open a  soft, or pseudo, keyboard: the Flex way, the Quick way and the Default way.

- The Flex way involve doing each step of setting up a soft keyboard separately. The keyboard is created using APIs such as SkyCreate( ), SkySetOrigin( ), SkySetKeySize( ), SkySetKeyMap( ), SkyBind ( ) and SkyOpen( ). This method of creation offers flexibility.
- The Quick way do all the functions with one API; SkyOpenKB( ). This method offers a quick way to open up a keyboard with flexibility of customising the keyboard.
- The default way, is the quickest way of open a default keyboard as shown in *Figure 23.1* using the SkyOpenDefKB ( ) API.

For all the 3 ways, when the soft keyboard is opened, PPSM GT will save the display area covered by the soft keyboard and monitors the input keys automatically. The soft keyboard is now ready for user's input.

When the user presses a key on the soft keyboard, the pre-defined ASCII code for that key is returned to the calling application by way of event when the application calls EvtCheck( ) or EvtGet( ).  One EVT_SKY_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.

## Creating the keyboard the Flex way

| | |
|---|---|
| STATUS **SkyCreate**(P_SKY_ID pSkyId) | Create softkey. Allocate memory for softkey structure. The softkey is not displayed on screen after created, user should configure the softkey properly by the following APIs. |
| STATUS **SkySetOrigin**( SKY_ID skyId, U16 xPos, U16 yPos) | Set the top-left hand corner of the softkey. |

| | |
|---|---|
| STATUS **SkySetKeySize**( SKY_ID skyId, U16 keyWt, U16 keyHt) | Set the width and height of each key on the softkey in number of pixels. |
| STATUS **SkySetKeyMap**( SKY_ID skyId, P_U16 keyMap, P_U8 keyBmp, U16 bmpWt, U16 bmpHt) | Set the keycode mapping and the bitmap of a softkey. The keyMap is a string of keycode, for example, "QWERTYUIOP...". The keyBmp is a pointer to the bitmap of the softkey. bmpWt and bmpHt is the width and height of the softkey. |
| STATUS **SkyBind**( SKY_ID skyId, APP_ID appId) | Bind the softkey to an application. The softkey bitmap is put on the panning screen of that application. |
| STATUS **SkyOpen**( SKY_ID skyId) | Open a softkey. Store background area then put the softkey bitmap on that area. |

## Opening the keyboard the Quick way

| | |
|---|---|
| STATUS **SkyOpenKB**(P_SKY_ID pSkyId, U16 xPos, U16 yPos, U16 keyWidth, U16 keyHeight, U16 numCol, U16 numRow, P_U16 keyMap, P_U8 bitmap) | Open a self-defined softkey with the given configuration. |

## Opening the keyboard the Default way

| | |
|---|---|
| STATUS **SkyOpenDefKB**(P_SKY_ID pSkyId, U16 xPos, U16 yPos) | Open a default softkey at the given position. |

## Auto-Key-Repeat

| | |
|---|---|
| STATUS **SkySetAutoRepeat**(SKY_ID skyId, U16 beginTime, U16 repeatTime) | • Set the auto repeat duration of softkey.<br>• Continually pressing a key would result in auto-key-repeat. The time between the first and second returned key can be set by the SkySetAutoRepeat( ) API. Two parameters are set using this API;<br>• BeginTime : Time between PEN touch and first repeat<br>• RepeatTime : duration between each repeat<br>• By default, the auto key repeat is enabled. The beginTime is 800 milliseconds and the repeatTime is 300 milliseconds. |

## Terminating Soft Keyboard Character Input

One API SkyClose( ) is used to close a softkey which is opened by either SkyOpen( ), SkyOpenKB( ), or SkyOpenDefKB( ).

PPSM GT will restore the display area that was covered by the soft keyboard automatically when the softkey is closed.

| | |
|---|---|
| STATUS **SkyClose(** SKY_ID skyId) | • Close a softkey as specified by the skyId.<br>• Clear softkey bitmap and put back the background area bitmap.. |

## Summary

PPSM-GT provides a default soft keybroad with three soft keyboard layouts: one for upper case letters(refer to *Figure 23.1*), one for lower case letters (refer to *Figure 23.2*), and one for numbers and symbols (refer to Figure 23.3). To use the default layouts or build customized versions of the keyboards, there are 3 ways provided; the Flex, Quick, and Default ways.

## Code Examples

### Listing 23.1 Open default soft keyboard for input

```
SKY_ID keyId;

/* open default soft keyboard for input */
      if ( SkyOpenDefKB(&keyId, KEYBD_X, KEYBD_Y)
!= SYS_OK )
        return (SYS_ERROR);
```

### Listing 23.2 Open self defined 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 ( SkyOpenKB(&NUMKBID, KEYBD_X, KEYBD_Y, 10, 10,
3, 4, (P_U16)keyMap, bitMap) != SYS_OK )
    return (SYS_ERROR);
```

24

# Pen Input Handling Services

The Pen input handling services are for handling inputs from the touch screen panel. This chapter covers information on the type of input area supported by PPSM-GT and only the way to handle them. Handwriting recognition input pad is a special type of input areas and is not covered in this chapter but in the Handwriting recognition chapter.

This chapter is organised into the following:

- Pen Input Handling Fundamental
- Programming using Pen Input Handling Services
- Summary

## Pen Input Handling Fundamental

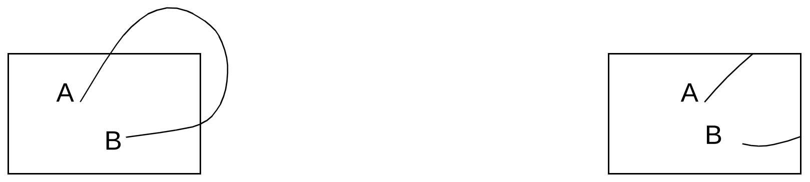
To understand the pen input handling service, it is important to be be familiar with some of the basic terms used in pen input handling. Table 24.1 shows the basic terminology used in pen input handling services.

**Table 24.1 Pen Handling Terminology**

| Term | Description |
|---|---|
| Active area | Area on the touch screen panel that takes movements when touched. |
| Touch panel coordinate | The x, y coordinate values directly come from touch panel controller. |
| Display coordinate | The x, y coordinate values respects to display screen in number of pixels. |
| Icon area | Active area that takes only pen up, pen down, pen drag in and out inputs only. Any pen movement within this area will not result in any responses. No echoing will be implemented. Each action like pen up, pen down, pen drag in and pen drag out will have separate event to be sent to the target task |
| Input area | Input area are active area that takes all the pen movements within the area. PPSM-GT supports input area with continuous input, stroke input and confined input |
| Continuous input | Continuous stream of x, y coordinates between a pair of pen-down and pen-up.<br><br>Input area where each sampling point will be sent to the target task with an individual event. Each action like pen up, pen down, pen drag in and pen drag out will have separate event to be sent to the target task |
| Stroke input | Input area where each stroke of sampling points will be sent at once to the target task at pen up or pen drag out within an event. Each action like pen up, pen down, pen drag in and pen drag out will have separate event to be sent to the target task |

| Term | Description |
|---|---|
| Confined input | It's similar to the stroke input except that the stroke of sampling time will be sent at once to the target task in pen up only. Whenever the pen drags out of the input area, it will be treated as writing on the boundary. So the pen outside the input area will be treated as still within the area as Figure 24.1 |
| Echo mode | Echo mode is only available in input area. If it's TRUE, whenever pen is touching that area, points will be drawn on LCD. If it's FALSE, nothing will be drawn on LCD even pen is touching the panel. |

**Figure 24.1 Confined input active area**



The pen draws the line in left diagram and the points returned will be a line starting from A towards B along the boundary. Each action like pen up and pen down will have separate event to be sent to the target task. Pen drag in and pen drag out will have no event sent to the target task

## Input Context

Input contexts are memory buffer that store pen input time-out, sampling rate, pen size, pen color, active area list, and task that is bound to the input context.

The input context data structure shown in Table 24.2 illustrate that input context setup the template for the pen input data.

**Table 24.2 Input Context Data Structure**

| Descriptions |
|---|
| Input Context Identifier |
| Application identifier where this input context is bound to |
| Task identifier where event will be sent |
| Pen input time-out start counting in pen up |
| Pen sampling rate in number of samples per second |
| Pen echoing size in number of pixels |
| Pen echoing color |
| Active area list |

- Pen Input Time-out controls the time-out period between pen up and the next pen down. This parameter is used by PPSM GT to determine whether, the input has been completed especially for hand writing recognition, where the input completed event needs to be sent to the hand writing recognition engine for recognition.
- Sampling rate determines the number of samples required per second.

## Active Area

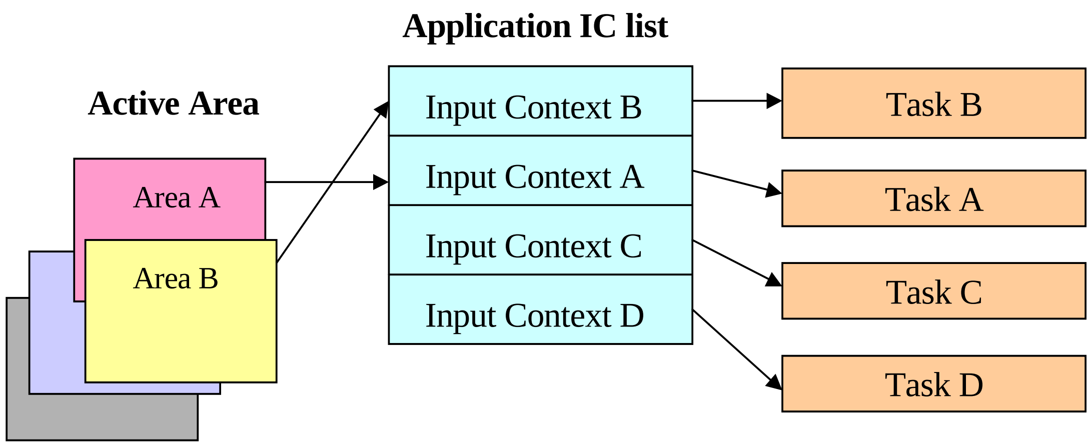
Active area provides an easy method for applications to receive pen input samples from the touch panel without the need to monitor the hardware constantly. PPSM GT uses interrupt to perform pen sampling, maximizing processor's utilization.

An active area is defined as a rectangular region of the touch panel where an event will be sent out by the system to the target task. The target task is bound to the input context before adding the input context to the application's input context list. An example of an active area is an icon, an action button, scratch pad or drawing area.

Active areas are only "active" when the input context containing the active area is bound to the current application. To deactivate active areas, they need to be removed from the input context or the whole input context containing the active area is removed from the input context list in the current application.

## Overlapping Active Area in different IC

**Figure 24.2 Overlapping active area in different IC**

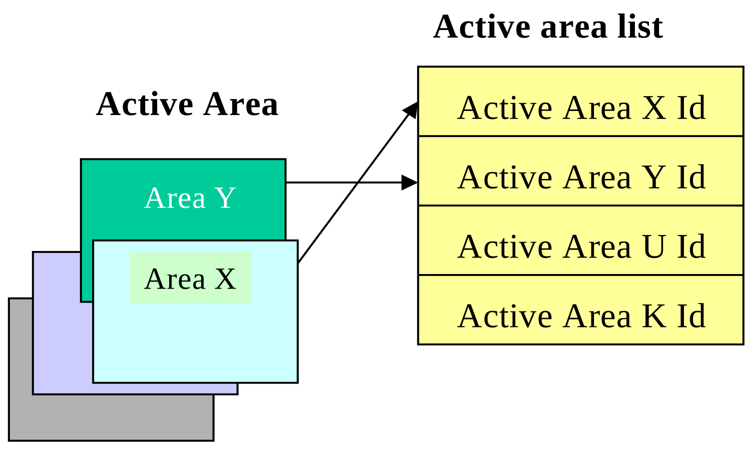


When active areas from different ICs are overlapping, then which task will receive the event informing the touch of active area will depend on the touched area position in input context and the position of the input context in the input context list of the current application. Figure 24.2 provides the illustration. Each active area belong to the different IC that are stored in the application's IC's list as shown.

- If area A is touched, then task A will receive the event as the area is not overlapped.
- If area B is touched, as it is an overlapped area, then task B will receive the event as it's position on the IC's list is higher than task A. Therefore, the position in the IC's list determines the priority of the task to receive the event when an overlapped active area is touched.
- When AppMoveICToTop() is called, it moves the specified IC to the top of the IC list hence making the specified IC the highest priority of the list.

## Overlapping Active Area in Same IC

**Figure 24.3 Overlapping active area in same IC**

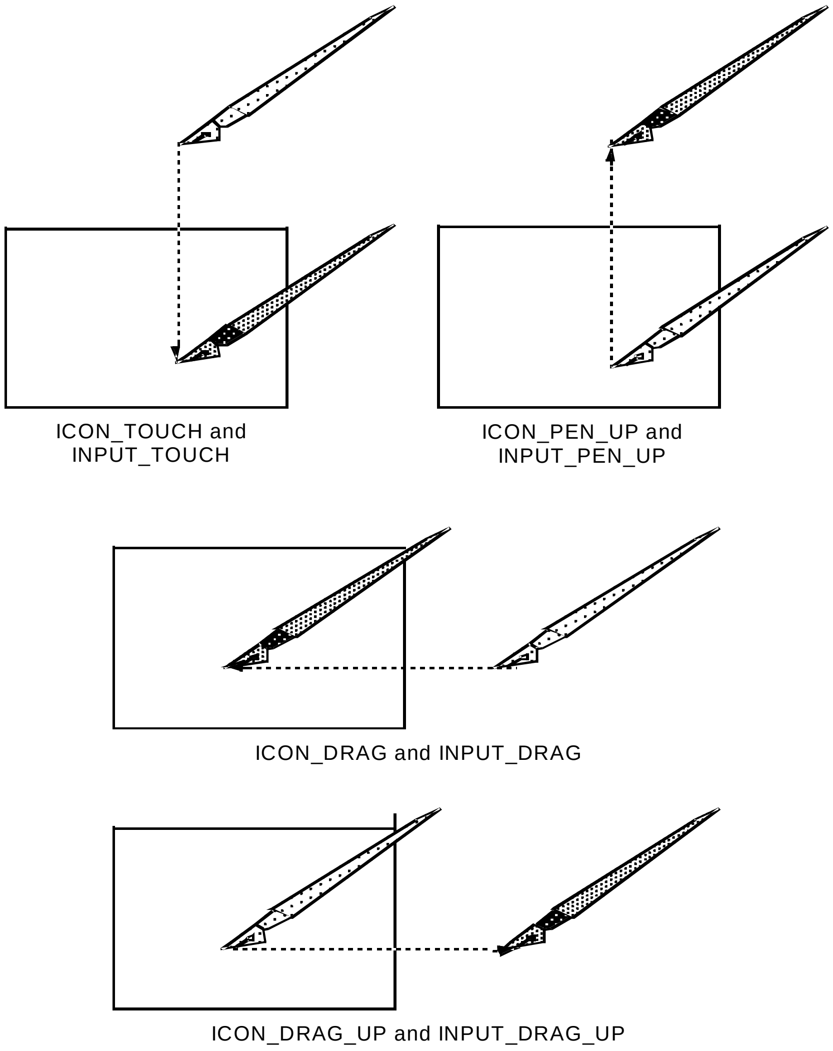


When active areas in the same IC are overlapping, then which active area Id will be sent to the task will depend on the position of the active area id. in the active area list of the input context. Figure 24.3 provides the illustration. Each active area has a different id. that is stored in the active area list as shown.

- If area Y is touched, then active area Y's id. will be sent as the area is not overlapped.
- If area X is touched, as it is an overlapped area, then active area X's id will be sent instead of Y as it's position on the active area list is higher than task Y. Therefore, the position in the active area list determines the priority of theactive area Id to be sent when an overlapped active area is touched.
- PenBringAreaBack( ), PenBringAreaBackward( ), PenBringAreaForward( ), PenBringAreaFront( ), PenMoveAreaToTop( ) are APIs provided for manipulating the active areas position in the active area link list. Therefore, they could be used to priortize the active area under active area overlapping situation.
- Whenever an active area is added to the input context, it will be put in the top position in the active area list inside the input context.

## Type of Active Area

**Figure 24.4 IDifferent type of Pen Input**



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

| Type | Mode | |
|---|---|---|
| ICON_AREA | N/A | Icon area has only one mode |

| Type | Mode | |
|---|---|---|
| INPUT_AREA | STROKE_MODE | Stroke input mode |
| | CONTINUOUS_MODE | Pen position sampling mode |
| | CONFINED_MODE | Strokes confined within the area |

## 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-GT will send an event to the target task that is on the input context containing the active area identifier.

Upon release, either by pen-up or drag out of the area into another part of the touch panel, another event will be sent to the target task to notify of the action.

This type of area is designed for buttons and selection icons.

## 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.

### Stroke Mode

Drawing on STROKE type of input area will produce a list of the x and y coordinate integers to the target task at the end of the drawing input sequence, usually with a pen-up or drag out. 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 and an event will be sent to the target task.

### 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. SO no pen drag out nor pen drag will be sent to the target task. The pen will always be within the active area once it’s touched or dragged in. On event will be sent to the target task with the whole stroke data upon pen up.

### Continuous Mode

Drawing on CONTINUOUS type of input area will continuously produce individual x and y coordinates to the target task as the pen moves across the pen input panel. With this type of input, event is generated for each individual point. Developers using this type of area must ensure the events are acknowledged as their number can be very significant.

## Relationship between active area, input context, task and application

**Figure 24.5 Relationship active area, IC, and tasks**

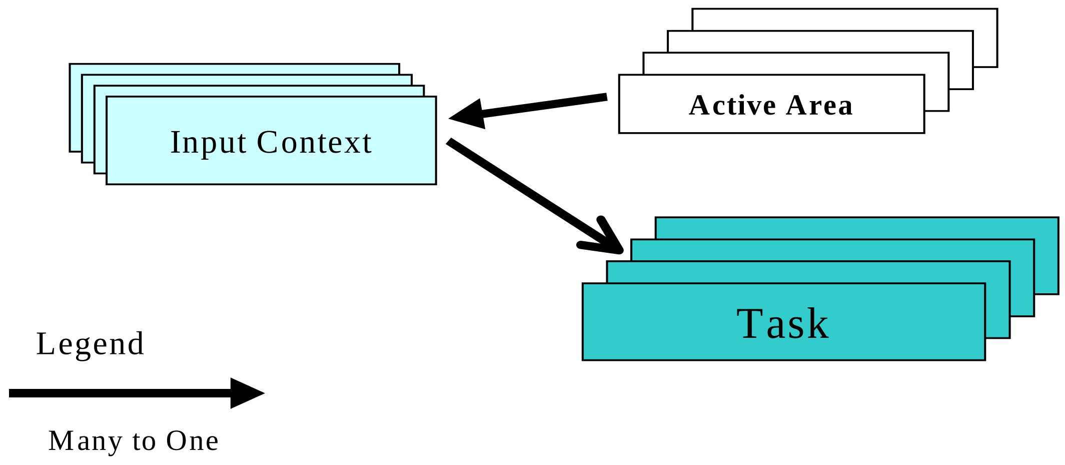


Figure 24.5 illustrates the relationship of active area, input context and tasks. One input context could have many active areas but the active area can only belong to one input context.

One task can also have many input contexts such that different type of input input areas could be found in the same task. However, each input context could only be bound to one task at any one time. This relationship could be changed with PenBindTaskToIC( ) API that binds the specified task to the input context.

## Input pad(for Handwriting recognition)

It is a task with specified priority. It will get the points when the pen is touching the input pad area. Then the data will be sent to the handwriting recognition engine from third party vendor. When the pen is up or the pen is moved to other active area or other input pad, the handwriting recognition engine will start to recognize the word. The candidates generated in the handwriting recognition task will be sent to the task creating this input pad.

# Programming using Pen Input Handling Services

The APIs in the pen input handling services are structured into 3 main areas: Input context manipulation, Active area control and Display and general setup.

## Creating Input Context

| | |
|---|---|
| STATUS **PenCreateIC**(P_IC_ID pIC) | Input contexts controls the input properties for the system. They work with application and have to be created in order for the system to receive input. PenCreateIC( ) creates and allocates memory for input context, and return ERR_MEM_NO for invalid memory pointer or no memory available or SYS_OK for successful operation |

## Initializing Input Context

| | |
|---|---|
| STATUS **PenInitIC**(IC_ID icId, TASK_ID taskId, TICK penInputTimeout, PEN_RATE samplingRate, U8 iconScan, U8 penSize, COLOR penColor) | The Pen Init( ) will initiate and set up the pen input handling data structure. The following are the options:<br>– Input Context Id is the IC for the input context<br>– Task id. for the task that will receive the message for any action happens inside the active area.<br>• Pen input time-out<br>• Pen sampling rate<br>– PEN_4HZ<br>– PEN_8HZ<br>– PEN_16HZ<br>– PEN_32HZ<br>– PEN_64HZ<br>– PEN_128HZ<br>– PEN_256HZ<br>– PEN_512HZ.<br>• Pen echo size in pixel<br>• Pen echo color<br>• Return<br>– ERR_MEM_NO for invalid memory pointer / no memory available or<br>– SYS_OK for successful operation |

## Modifying Pen Input Parameter

PenBindTaskToIC( ), PenSetInputTimeout( ), PenSetPenColor( ), Pen SetPenSize( ), PenSetEchoMode(), and PenSetSamplingRate( ) are APIs provided to increase the flexibility of modifying the input context without deleting and recreating the IC. PenBindTaskToIC( ) offers further flexibility for task to use and reuse IC when necessary. However, the rule of one IC to one task at any one time have to be observed.

| | |
|---|---|
| STATUS **PenBindTaskToIC**(TASK_ID taskId, IC_ID icId) | PenBindTaskToIC( ) adds the task id. into the input context so those event for this input context will be sent to the specific task, and Return<br>– ERR_APP_TASK_ID for invalid task id,<br>– ERR_APP_IC_ID for invalid input context id, and<br>– SYS_OK for successful operation |
| STATUS **PenSetInputTimeout**(IC_ID icId, TICK time) | PenSetInputTimeout( ) sets the pen input time-out for the specific input context, and Return<br>– ERR_APP_IC_ID for invalid input context id, or<br>– SYS_OK for successful operation |

| | |
|---|---|
| STATUS **PenSetPenColor**(IC_ID icId, COLOR penColor) | PenSetPenColor( ) sets the pen echo color for the specific input context, and<br>Return<br>– ERR_APP_IC_ID for invalid input context Id or<br>– SYS_OK for successful operation. |
| STATUS **PenSetEchoMode**(AREA_ID areaId, U8 echoMode) | PenSetEchoMode( ) sets the pen echo mode of the active area.<br>Return<br>– ERR_PEN_AREA_ID for invalid active area id, or<br>– SYS_OK for successful operation |

| | |
|---|---|
| STATUS **PenSetPenSize**(IC_ID icId, U8 penSize) | Pen SetPenSize( ) sets the pen echo size in pixel for the specific input context, and Return<br>– ERR_APP_IC_ID for invalid input context Id or<br>– SYS_OK for successful operation. |
| STATUS **PenSetSamplingRate**(IC_ID icId, PEN_RATE rate) | PenSetSamplingRate( ) sets the pen sampling rate for the specific input context to:<br>– PEN_4HZ<br>– PEN_8HZ<br>– PEN_16HZ<br>– PEN_32HZ<br>– PEN_64HZ<br>– PEN_128HZ<br>– PEN_256HZ<br>– PEN_512HZ.<br>Return<br>– ERR_APP_IC_ID for invalid input context Id or<br>– SYS_OK for successful operation. |

## Creating Active area

| | |
|---|---|
| STATUS **PenCreateArea(**P_AREA_ID pAreaId) | Active areas are area on the touch panel that are setup to receive input. They work with input context and are specified the type of input to be received. PenCreateArea( ) allocates the memory for active area and return the active area id. The error message ERR_MEM_NO is returned for invalid memory pointer or no memory available or SYS_OK for successful operation. Active area will not be active after creation unless it's added to the input context which is inside the input context list in current application. |

## Initializing active area

| | |
|---|---|
| STATUS **PenInitArea**(AREA_ID areaId, S16 xSrc, S16 ySrc, S16 xDest, S16 yDest, U8 type, U8 mode, U8 panPosFlag, U8 echoMode) | PenInitArea( ) will set up the created active area ready to be used as followed:<br>• areaId : Active area to initialize<br>• xSrc : Top left x coordinate<br>• ySrc : Top left y coordinate<br>• xDest : Bottom right x coordinate<br>• yDest : Bottom right y coordinate<br>• type : ICON_AREA, INPUT_AREA or KEY_AREA<br>• mode : CONTINUOUS_MODE, STROKE_MODE or CONFINED_MODE<br>• panPosFlag : TRUE(use panning screen coordinate for the active area), FALSE(use LCD display coordinate for the active area).<br>• echoMode : TRUE echo input else no echo<br>• Return<br>– ERR_PEN_AREA_ID for invalid active area id.or<br>– SYS_OK for successful operation. |

It’s recommended not to use panning screen coordinate in confined area as the returned pen coordinate will always be LCD display coordinate.

## Adding Pen Input Area To IC

| | |
|---|---|
| STATUS **PenAddAreaToIC**(IC_ID icId, AREA_ID areaId) | PenAddAreaToIC( ) adds an active area to an input context, and return ERR_APP_IC_ID for invalid input context id. ERR_PEN_AREA_ID for invalid active area id. ERR_APP_AREA for active area already added to other input context, and SYS_OK for successful operation |

## Removing Pen Input Area From IC

| | |
|---|---|
| STATUS **PenRemoveAreaFromIC**(AREA_ID areaId) | PenRemoveAreaFromIC( ) API allow the removing of active area from the IC so that when active area is not required for any situation, they could be suspended and not deleted, i.e it does not free up the memory.<br>This API will eliminate the need to delete and recreate area when needed. To use the active are again use PenAddAreaToIC( ).<br><br>Return<br>– ERR_APP_IC_ID for invalid input context or<br>– SYS_OK for successful operation |

## Modifying Active Area parameters

PenSetAreaMode( ), PenSetAreaPos( ), and PenSetAreaType( ) are APIs for modifying the parameters in the active area data structure. They are provided to eliminate the need to delete and recreate the active if there is a change in position, type or mode. The same active area could be reused by modifying the affected parameters with the following APIs:

| | |
|---|---|
| STATUS **PenSetAreaMode**(AREA_ID areaId, AREA_MODE mode) | PenSetAreaMode( ) sets the mode into the active area to CONTINUOUS_MODE, STROKE_MODE or CONFINED_MODE. Return<br>– ERR_PEN_AREA_ID for invalid active area id, or<br>– SYS_OK for successful operation |

| | |
|---|---|
| STATUS **PenSetAreaPos**(AREA_ID areaId, S16 xSrc, S16 ySrc, S16 xDest, S16 yDest) | PenSetAreaPos( ) sets the active area coordinates in the specific active area.<br>Return<br>– ERR_PEN_AREA_ID for invalid active area id, or<br>– SYS_OK for successful operation |
| STATUS **PenSetAreaType**(AREA_ID areaId, AREA_TYPE type) | PenSetAreaType( ) sets the active area type for the specific active area to ICON_AREA, INPUT_AREA or KEY_AREA(reserved for softkeyboard).<br>Return<br>– ERR_PEN_AREA_ID for invalid active area id, or<br>– SYS_OK for successful operation |

## Changing Active Area In Area Link List

PenBringAreaBack( ), PenBringAreaBackward( ), PenBringAreaForward( ), PenBringAreaFront( ), PenMoveAreaToTop( ) are APIs provided for manipulating the active areas position in the active area list. In overlapping active area situation, these APIs would be useful to bring active area to front or back depending on the requirement.

| | |
|---|---|
| STATUS **PenBringAreaBack**(AREA_ID areaId) | PenBringAreaBack( ) moves the active area to tail of the active area list in the input context it belongs to. Return<br>– ERR_PEN_AREA_ID for invalid active area id,<br>– ERR_APP_IC_ID for Active area doesn't belong to a valid input context, and<br>– SYS_OK for successful operation |
| STATUS **PenBringAreaBackward**(AREA_ ID areaId) | PenBringAreaBackward( ) moves the active area one step towards the tail of the active area list in input context. Return<br>– ERR_PEN_AREA_ID for invalid active area id, or<br>– SYS_OK for successful operation |
| STATUS **PenBringAreaForward**(AREA_I D areaId) | PenBringAreaForward( ) moves the active area one step towards the head of the active area list in input context. Return<br>– ERR_PEN_AREA_ID for invalid active area id, or<br>– SYS_OK for successful operation. |

| | |
|---|---|
| STATUS **PenBringAreaFront**(AREA_ID areaId) | PenBringAreaFront( ) moves the active area to the front of the active area list in the same input context.<br>Return<br>– ERR_PEN_AREA_ID for invalid active area id, or<br>– SYS_OK for successful operation |
| STATUS **PenMoveAreaToTop**(IC_ID icId, AREA_ID areaId) | PenMoveAreaToTop( ) moves the active area to the top of the active area list of the specified input context.<br>Return<br>– ERR_APP_IC_ID for invalid input context id,<br>– ERR_PEN_AREA_ID for invalid active area id,<br>– ERR_APP_AREA for the area is not in the input context, and<br>– SYS_OK for successful operation |

## Deleting Active Area

| | |
|---|---|
| STATUS **PenDeleteArea**(AREA_ID areaId) | PenDeleteArea( ) API is used to delete active area that are no longer in used. It free the active area memory and remove the active area from input context if it's attached to any input context.<br>Return<br>– ERR_PEN_AREA_ID for invalid active area id, or<br>– SYS_OK for successful operation |

## Display and General Setup

### Enabling Pen Calibration

| | |
|---|---|
| STATUS **PenCalibrate**(U8 logoFlag) | Align the input device withthe LCD display. Also displayed Motorola logo and the crosses for pen calibration.<br>Boolean input :<br>• TRUE for calibration and<br>• FALSE no calibration. |

### Mapping Touch Screen Coordinates

| | |
|---|---|
| U16 **PenMapX**(U16 x)<br>U16 **PenMapY**(U16 y) | PenMapX( ) converts the touch panel x coordinate to LCD display x coordinate.<br>PenMapY( ) converts the touch panel y coordinate to LCD display y coordinate |

### Setting up the Ring Buffer

| | |
|---|---|
| STATUS **PenSetRingBuffer**(APP_ID appId, U16 bufferSize) | PenSetRingBuffer( ) sets the ring buffer size for saving pen sampling data temporarily.<br>Return<br>– ERR_APP_ID for invalid application id, or<br>– SYS_OK for successful operation |

## Getting IC from Active Area

| | |
|---|---|
| STATUS **PenGetICFromArea**(AREA_ID areaId, P_IC_ID pIcId) | PenGetICFromArea returns the input context that is bind to the active area list, and<br>Return<br>– ERR_MEM_NO for invalid memory pointer or<br>– SYS_OK for successful operation. |

## Getting AreaId From Event

| | |
|---|---|
| STATUS **PenGetAreaIdFromEvent**(P_EVENT pEvent, P_AREA_ID pAreaId) | PenGetAreaIdFromEvent( ) gets the active area id. from the event message sent from pen task.<br>Return<br>– ERR_MEM_NO for invalid memory pointer, or<br>– SYS_OK for successful operation |

## Getting Area Mode

| | |
|---|---|
| STATUS **PenGetAreaMode**(AREA_ID areaId, P_AREA_MODE pMode) | PenGetAreaMode( ) returns the active area mode for the specified active area id. Return<br>– ERR_MEM_NO for invalid memory pointer,<br>– ERR_PEN_AREA_ID for invalid active area id, and<br>– SYS_OK for successful operation |

## Getting Active Area Position

| | |
|---|---|
| STATUS **PenGetAreaPos**(AREA_ID areaId, P_S16 pXSrc, P_S16 pYSrc, P_S16 pXDest, P_S16 pYDest) | PenGetAreaPos( ) returns the active area position for the specified active area id. Return<br>– ERR_MEM_NO for invalid memory pointer,<br>– ERR_PEN_AREA_ID for invalid active area id, and<br>– SYS_OK for successful operation |

## Getting Active Area Type

| | |
|---|---|
| STATUS **PenGetAreaType**(AREA_ID areaId, P_AREA_TYPE pType) | PenGetAreaType( ) returns the active area type for the specified active area id. Return<br>– ERR_MEM_NO for invalid memory pointer,<br>– ERR_PEN_AREA_ID for invalid active area id, and<br>– SYS_OK for successful operation |

## Getting Pen Info

| | |
|---|---|
| STATUS **PenGetEchoMode**(AREA_ID areaId, P_U8 pEchoMode) | PenGetEchoMode( ) returns the echo mode for the specified active area id.<br><br>Return<br>– ERR_MEM_NO for invalid memory pointer,<br>– ERR_PEN_AREA_ID for invalid active area id, and<br>– SYS_OK for successful operation |
| STATUS **PenGetInputTimeout**(IC_ID icId, P_TICK pInputTimeout) | PenGetInputTimeout( ) returns the pen input time-out for the specific input context. The input time-out is the time between the pen up event and the interrupt to signal the time-out. So if the pen touches the panel within this time-out length, no event will be sent to the task linked to the input context.<br><br>Return<br>– ERR_MEM_NO for invalid memory pointer,<br>– ERR_APP_IC_ID for invalid input context id, and<br>– SYS_OK for successful operation |

| | |
|---|---|
| STATUS **PenGetPenColor(**IC_ID icId, P_COLOR pPenColor) | PenGetPenColor( ) returns the pen echo color for the specific input context.<br><br>Return<br>– ERR_MEM_NO for invalid memory pointer,<br>– ERR_APP_IC_ID for invalid input context id, and<br>– SYS_OK for successful operation. |
| STATUS **PenGetPenSize**(IC_ID icId, P_U8 pPenSize) | PenGetPenSize( ) returns the pen echo size in pixels for the specific input context.<br><br>Return<br>– ERR_MEM_NO for invalid memory pointer,<br>– ERR_APP_IC_ID for invalid input context id, and<br>– SYS_OK for successful operation. |

| | |
|---|---|
| STATUS **PenGetPos**( P_S16 pX, P_S16 pY) | PenGetPos( ) returns the current pen coordinate in LCD coordinate where the top left hand corner of the LCD is always (0, 0).<br><br>Return<br>– ERR_MEM_NO for invalid memory pointer, or<br>– SYS_OK for successful operation. |
| STATUS **PenGetPosFromEvent**( P_EVENT pEvent, P_POINT *pPoints, P_U16 pNumberOfPoints) | PenGetPosFromEvent( ) returns the stroke data and the number of points in the stroke from the event sent from pen task.<br><br>Return<br>– ERR_MEM_NO for invalid memory pointer, or<br>– SYS_OK for successful operation. |

| | |
|---|---|
| VOID **PenGetSample**( ) | PenGetSample( ) is supposed to be called by sampling timer to get the pen sample points periodically. User may have different periodic timer and this can be called by the user defined periodic timer to send pen sample point to pen task |
| STATUS **PenGetSamplingRate**(IC_ID icId, P_PEN_RATE pPenRate) | PenGetSamplingRate( ) returns the pen sampling rate for the specific input context.<br><br>Return<br>– ERR_MEM_NO for invalid memory pointer,<br>– ERR_APP_IC_ID for invalid input context id, and<br>– SYS_OK for successful operation. |

## Summary

This chapter provides information on the input context and the active area. Input context are basically memory buffers that stored the information on pen input time-out, sampling rate, pen size in number of pixels, pen color, active area list, and task ID. All this information provide the input parameters and the echo characteristics of the pen input. Active Areas are areas on the touch screen panel that provide an easy method for application to receive pen input sample. APIs are also provide to setup the characteristics of the active area to enable different type of pen inputs.

# 25

# Handwriting Recognition Input Handling Services

PPSM-GT supports input pad that could be used as handwriting recognition input pad. When enabled, the system will capture all inputs stroke between pen down and pen up, and send the inputs to the handwriting engine for recognition.

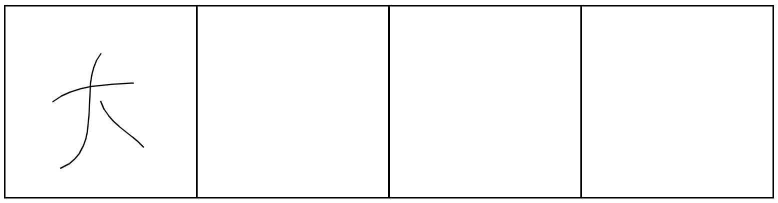
The system also handle the saving and restoring of the background image that was covered by the input pad, when the input pad is closed.

This chapter is organised as followed:

## Handwriting Recognition Input Fundamental

The handwriting recognition services are essentially a group of specially defined input pads designed for capturing input data for pen stroke inputs. It consists of a number of square boxes in a row by column format layout (refer to *Figure 25.1*).

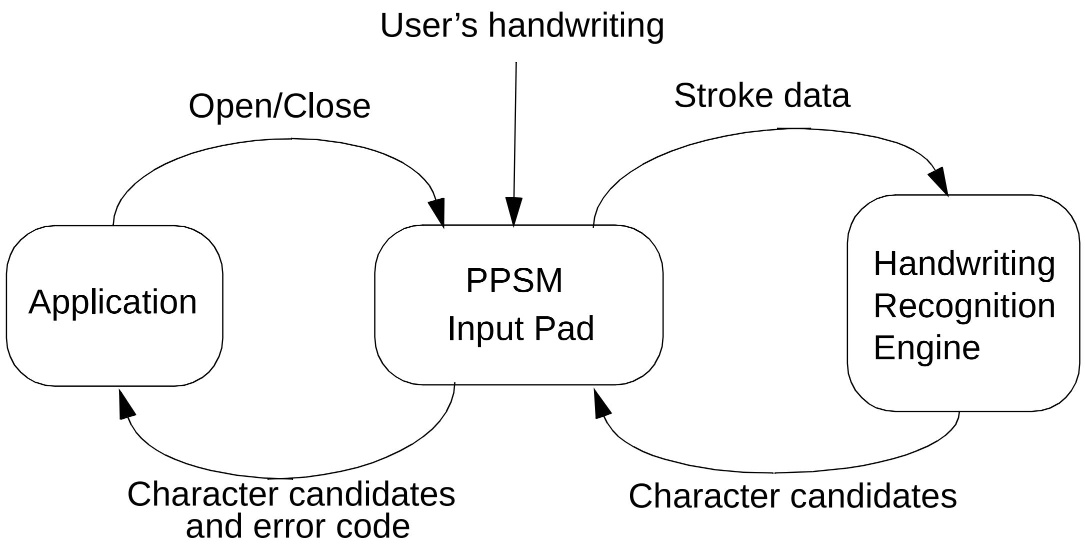
**Figure 25.1 An Example Input Pad with 1 row by 4 column layout**



It serves as an interface between the user and the underlying handwriting recognition engine. It captures the stroke data generated from the user's handwriting input, and passes these data to the handwriting recognition engine for processing. (Refer to *Figure 25.2* for the flow of input and output data passing through the input pad).

PPSM-GT provides APIs for installing and uninstalling handwriting engine into PPSM-GT system. When the handwriting engine is installed, any hand writing input will be sent to the HWR engine for recognition.

**Figure 25.2 Data flow of Handwriting Recognition Input Pad**



## 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).

Only ONE instance of the input pad is supported per main task. If the user opened an input pad in one of the tasks, an attempt to open another the input pad would fail.

# Programming using Input Pad Handling Services

## Opening Handwriting Character Input

| | |
|---|---|
| STATUS **InpOpen**(P_INP_ID pId, HWR_ID hwrId, TASK_ID taskId, U16 xPos, U16 yPos, U16 numRow, U16 numCol, U16 areaWidth, U16 areaHeight, TICK timeout, U8 echoSize) | An application can open the input pad anywhere within the panning screen. The application needs to specify the Pointer to input pad ID, HWR ID, 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 width and height (in units of pixel), timeout(no more than 1sec) and ink echo size.<br>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's input.<br>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.<br>The result will be send to the task specified by the task Id. |

## Accessing a general Input Pad

| | |
|---|---|
| STATUS **InpDrawPad**(INP_ID inpId) | InpDrawPad( ) draws an input pad on the screen. It also enable the automatic clear and restore the background image after the input pad being closed |

## Setting Handwriting Input pad Sampling Rate

| | |
|---|---|
| STATUS **InpSetSamplingRate**(INP_ID id, PEN_RATE time) | The default length of sampling rate for InpOpen( ) is 1 second. To change the sampling rate InpSetSamplingRate( ) will set the specified sampling rate. |

## Setting the Pen Echo Color

| | |
|---|---|
| STATUS **InpSetColor**(INP_ID id, COLOR color) | The default ink echo color for InpOpen( ) is black. To change the color InpSetColor( ) will set the specified color. |

## Terminating Handwriting Character Input

| | |
|---|---|
| STATUS **InpClose**(INP_ID id) | InpClose( ) closes the input pad that has been opened either by InpOpen( ) or InpDrawPad( ). 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. |

## Installing HWR engine

| | |
|---|---|
| STATUS **InpInstallHWR**(P_HWR_ID pId, P_VOID resetEng, P_VOID initEng, P_VOID processStk, P_VOID recgzInp, U32 stackSize) | InpInstallHWR( ) is for installing 3rd party hand writing engine into PPSM-GT system. The input parameters are pointer to HWR ID, and pointer to HWR engine functions and stack size. The stack size is for the proper operation of the HWR engine. |

## Reading HWR engine.

| | |
|---|---|
| STATUS **InpGetCandidates**(P_TEXT* pCandidates, P_U16 pNum, P_INP_ID pInpId) | It gets the HWR candidates from input pad and event |

## Uninstalling HWR engine

| | |
|---|---|
| STATUS **InpUninstallHWR**(HWR_ID id) | It uninstalls a hand writing recognization engine in PPSM |

## Checking HWR engine

| | |
|---|---|
| U8 **InpIsHWRId**(HWR_ID id) | It tests if a number is a HWR engine ID |

## Checking Input Pad

| | |
|---|---|
| U8 **InpIsInpId(**INP_ID id) | It tests if a number is an input pad ID |

## Bring the input pad to the top of the IC

| | |
|---|---|
| STATUS **InpTop**(INP_ID id) | It tops an input pad on all the IC in the application |

## Summary

The PPSM-GT handwriting recognition services provide a meant to capture the pen stroke input and tranfer the data to the hand writing recognition engine for recognition. The PPSM-GT HWR services acts as a interface between HWR pads and the HWR engine. PPSM-GT also provides API to handle the HWR engine.

# Section 7

# Appendixes

This section contains the following appendixes:

# A

# Coding Conventions

The intent of this document is to relay to the user the coding conventions used to write PPSM_GT program. Most of the PPSM-GT routines are written in C and only a small portion of hardware dependence routines are written in 68K assembly language.

The goal of these conventions is to provide for consistent layout and formatting throughout all PPSM-GT code. Consistency makes the code easier to document and, most importantly, easier to scan.

## The Importance of Consistency

These conventions were not conceived to make life difficult for PPSM-GT programmers; their goal is to make it easier for customers to read the code. Experience teaches us that densely packed code with no consistent formatting inhibits understanding because the reader can never see a pattern in how the code is laid out. Thus, the code is hard to read.

With frameworks, the user often needs to be able to read the code in order to understand how a given feature works, as it is unrealistic to expect the documentation to cover every conceivable topic. Therefore, it is important to have consistent coding throughout the framework.

These coding conventions do not cover every aspect of coding with C, and purposefully avoid some of the more religious conventions in favor of conventions that are more layout oriented.

However, there are some general conventions regarding C presented here that are targeted at avoiding common pitfalls with C, as well as issues that might result in wasted space.

## Fonts

Code should always use a monospace font. The examples shown in this document use Courier New 12 point.

## Tabs and Spaces

In PPSM-GT code, a tab is equivalent to an indent. Tabs are never used to generate spaces within a line. A tab is equivalent to three spaces.

Where a tab is used as the first indent for a line, any subsequent horizontal alignment that takes place within the line is done with spaces. This convention allows alignment to work properly with different tab widths.

## Naming Conventions

This section outlines the naming conventions for elements like classes, constants, and globals.

The basic naming convention follows the form where the first character of each significant word is capitalized. For example:

- `KnlCreateTask`
- `AppCreate`

### Labels

All labels are in upper case, underscores are permitted. For example:

- DISPLAY_MODE
- DEFAULT_MODE
- OK
- UNKNOWN

## Local Variables

All local variables start with lower case, with capitalized words in the variable names. There are no underscores between words. For example:

- xPos
- yPos
- dataLen
- temp
- rate

## Global Variables

Global variables are prefaced with a lowercase “g” followed by a name conforming to the standard convention. For example:

- gCurrentTask
- gIrptMask
- gSystemClock

## Local Pointer Variables

All local pointer variables start with a lower case “p”, with capitalized words in the variable names. There are no underscores between words. For example:

- pSourceAddr
- pDestAddr
- pTaskTable
- pReturnSize

## Global Pointer Variables

Same as local pointer variables, except that all global variable names start with a lower case “gp”. For example:

- gpSysList
- gpPhoneBook
- gpSrcMem

## Local Variables

Local variable names always start with the first character of the first word in lowercase. Then, the first character of each significant word in the name is capitalized. For example:

- `arrowWidth = 20;`
- `arrowRect(0, 0, 20, 20);`

## Function and Method Names

Function and method names follow the normal naming convention; the first character of each significant word capitalized. This is shown in Listing 25.1. For example:

- PenInit()
- PagerCheck()
- DrawDot()

**Listing 25.1 Example function names**

```
Void PagerCheck()
{
          U8 numOfMesage;
          U8 lengthOfMessage;

CheckMessage(numOfMessage, lengthOfMessage);
}
```

## Parameters

Parameters follow the same naming conventions as local variables in that the first character is lowercase, followed by capitalizing the first character of each significant word in the name. Listing 25.2 shows examples of how parameters have been formatted.

**Listing 25.2 Example parameters**

```
CheckMessage(
          U8 numOfMessage,
          U8 lengthOfMessage);
```

## Macros

Macros are always all uppercase with an underscore ( _ ) separating each significant word. For example:

- FOUR_CHAR_CODE
- DEFINE_DERIVED_COMPONENT
- NET_COMPONENT

## Comments

There can never be enough code comments in a program. Within the body of a method or function, comments appear above the line of code being documented. These comments are typically done using the double slash ( // ) comment style. The primary reason for placing comments above the line as opposed to alongside is that it is highly annoying to have to continually scroll horizontally in order to read the comment.

Short comments use the // form, while long multi-line comments use the
/*…*/ form. Examples of this convention are shown in Listing 25.3.

**Listing 25.3 Example code comments**

```
// build a rectangle that represents the local frame of the view
frame(0, 0, GetWidth(), GetHeight());

/*
      This is a really long comment that spans multiple lines
      and therefore has been wrapped with the alternate comment
      form. NOTE: This not a requirement for long comments. The
      normal comment style is also used from time to time.
*/
```

## The Implementation's Identifier Space is respected

The PPSM-GT has reserved all identifiers that begin with an underscore ( _ ). This includes macros, local variable names, classes, function names, and more.

An identifier never starts with an underscore. A few include guards currently violate this standard, so take special care when dealing with them.

# File Layout

In order to make the layout of files easier, a set of stationery files is provided. These have the basic layout already set up, and can be used to create new files with the desired layout.

Where ever possible, the width of a line of code has been kept reasonable. This minimizes the amount of horizontal scrolling necessary to read the entire line. It is far better to wrap the line so that it is all visible than to have it disappearing of the right edge of the window.

This is also important when printing a file. By keeping lines short, code is not lost off the edge of the page.

The headers provided in the sample files have been laid out to fit a page. They can then be used as guides as to when a line is reaching a width that would be greater than a typical page.

## File Headers

A header is placed at the top of both the header and implementation files and contains the Motorola Semiconductor Hong Kong Limited copyright notice.Additionally, the header contains the name of the file, objective of the file, the creation date, the modification date, and—optionally—the author.

Separator lines have been laid out in the header to indicate the typical width of the printed page. This can then be used to set the width of a window in order to provide a guide as to where the lines wrap. Listing 25.4 shows a typical PPSM-GT header.

**Listing 25.4 A typical file header**

```
/****************************************************************
(c) copyright Motorola Semiconductor Hong Kong Limited 1998-2001
ALL RIGHTS RESERVED.

This example is the sole property of Motorola Inc. This software
cannot be distributed in whole or in part. This software is
provided as is and in no event shall Motorola, Inc. be liable for
any direct, special, incidental, or consequential damages arising
out of or connected with a users posession or use of this software
package, even if Motorola has advance notice of the possibility of
such damages.
****************************************************************/

File Name:grphDemo.c

Initial Creation Date: 990825

Modification Date:  $Date:$

=================================================================
Objective:

This section can be used to describe what this file contains and
what the code is used for, as well as any other useful
information.  The implementation file should typically contain a
detailed Theory of Operation description, as it makes life easier
for the user.
*
**/
```

**NOTE** The separators in the example above are much shorter than would typically be used in a source file due to the formatting of this document.

# B

# Error Message Handling

There are a few different ways PPSM-GT handles it’s error messages and each required a different way to access the error messages. This chapter will cover the types of error messages and how to get them.

**NOTE** Developers are encouraged to include error checking code in their program to capture and address improper system preformance.

The following are the different ways of handling error messages:

- PPSM-GT Core Error Handling
- Socket Services Additional Error Handling

## PPSM-GT Core Error Handling

In all the PPSM-GT core services except for networking services, the error messages are returned with each API call. The way to get them is to include the variable “status” to store the returned error code and checking it to take neccessary action as shown in code example in Error Checking for PPSM-GT core services“Error Checking for PPSM-GT core services,”

**Listing B.1 Error Checking for PPSM-GT core services**

```
status = MemResetRegion(regionId);

    if( status == SYS_OK)
    {
      sprintf(tempStrN, "Reset Success\n");
        else
```

```
sprintf(tempStrN, "Reset Fail\n);
```

Table B.1 shows all the possible error codes in PPSM-GT cores services. To get the definition of the error messages, please refer to the calling APIs in the PPSM-GT API reference manual. For example if when using the KnlCreateTask() API and the SYS_ERR error message is encountered, then please refer to the KnlCreateTask() API for the description of the error message.

The error messages are organized in such a way that they are specific to the calling API. The same error messages meant different things for the specific API. This is especially true for system message such as SYS_OK and SYS_ERR. They are used very often throughout the PPSM-GT design. A SYS_ERR message in RtcSetTime() API means the time setting is invalid whereas the same SYS_ERR error message in AudPlayMelody() API means that the melody music count is zero. It is therefore unwise to generalize. Please refer to the individual API for the definition of the error message when you encountered one.

**Table B.1 PPSM-GT Error messages and Definitions**

| Error Messages |
|---|
| **System Error Messages** |
| SYS_OK |
| SYS_ERR |
| **Kernel services error message** |
| ERR_KNL_TASK_ID |
| ERR_KNL_NO_MEMORY |
| ERR_KNL_IN_IRPT |
| ERR_KNL_PRIORITY |
| ERR_KNL_SEMA_ID |
| ERR_KNL_SEMA_TASK |
| ERR_KNL_SEMA_INIT |
| ERR_KNL_SEMA_COUNT |
| ERR_KNL_SEMA_TIME |
| ERR_KNL_SUSPEND_LV |
| ERR_KNL_STACK |
| ERR_KNL_SWAP_DISABLE |
| ERR_KNL_REG_ID |
| ERR_KNL_TASK_MODE |
| **Event Management services error messages** |
| ERR_EVT_INVALID |
| ERR_EVT_CHANNEL_ID |
| ERR_EVT_BROADCAST_ID |
| ERR_EVT_PORT_ID |
| ERR_EVT_NO_PORT |
| ERR_EVT_ADD_TASK |
| ERR_EVT_EVENT_INUSE |
| ERR_EVT_BROADCAST_INUSE |
| ERR_EVT_CHANNEL_ELM |
| ERR_EVT_TASK_NOTFOUND |

| Error Messages |
|---|
| ERR_EVT_BRDCST_NO_EVENT |
| ERR_EVT_BRDCST_LIMIT |
| ERR_EVT_TYPE |
| ERR_EVT_TIMEOUT |
| **Memory management services error messages** |
| ERR_MEM_NO |
| ERR_MEM_REGION_ID |
| ERR_MEM_CREATE_REGION |
| ERR_MEM_DATA_SIZE |
| ERR_MEM_INVALID_ADDR |
| **Alarm services error messages** |
| ERR_ALM_NO |
| ERR_ALM_PERIOD |
| **RTC handling services error  messages** |
| ERR_RTC_SECOND |
| ERR_RTC_MINUTE |
| ERR_RTC_HOUR |
| ERR_RTC_DAY |
| ERR_RTC_MONTH |
| ERR_RTC_YEAR |
| ERR_RTC_TMOUT |
| ERR_RTC_INIT |
| **SCI management Services Error Messages** |
| ERR_SCI_INVALID_ACCESS |
| ERR_SCI_INVALID_TMOUT |
| ERR_SCI_MODE |
| ERR_SCI_BAUD |
| ERR_SCI_PARITY |
| ERR_SCI_STOPBIT |
| ERR_SCI_CHARLEN |

| Error Messages |
|---|
| ERR_SCI_BUSY |
| ERR_SCI_FLAG |
| ERR_SCI_NO_REQUEST |
| ERR_SCI_INVALID_TXDELAY |
| ERR_SCI_PORT_ID |
| ERR_SCI_PORT_INUSE |
| ERR_SCI_RX_ON |
| ERR_SCI_TX_ON |
| ERR_SCI_NO_CTRL |
| ERR_SCI_NOT_BIND |
| ERR_SCI_DELAY |
| ERR_SCI_TX_BUSY |
| ERR_SCI_RX_BUSY |
| ERR_SCI_NO_RX |
| ERR_SCI_NO_TX |
| ERR_SCI_TASK_ID |
| ERR_SCI_RX_TMOUT |
| ERR_SCI_TX_TMOUT |
| ERR_SCI_RX_FIFO_LVL |
| ERR_SCI_TX_FIFO_LVL |
| ERR_SCI_UART_INUSE |
| ERR_SCI_UART_PORT |
| ERR_SCI_FIFO_LEVEL |
| **Audio handling services error messages** |
| ERR_AUD_INUSE |
| ERR_AUD_SAM |
| ERR_AUD_REGS |
| ERR_AUD_TONEDUR |
| ERR_AUD_TONEVOL |
| **Power management services error messages** |

| Error Messages |
|---|
| ERR_PWR_MODE |
| ERR_PWR_IDLE_LIMIT |
| ERR_PWR_IDLE_DISABLE |
| **Software Timer handling services error messages** |
| ERR_SWT_ID |
| ERR_SWT_TIME_LIMIT |
| ERR_SWT_POINTER |
| **Application services error messages** |
| ERR_APP_IC_ID |
| ERR_APP_AREA |
| ERR_APP_ID |
| ERR_APP_PAN_INFO |
| ERR_APP_TASK_ID |
| **Interrupt service routine error messages** |
| ERR_ISR_UNDEF |
| ERR_ISR_OCCUPIED |
| ERR_ISR_LV_RANGE |
| ERR_ISR_LV_UNCONFIG |
| **IrDA services error messages** |
| ERR_IRD_STATUS_SUCCESS |
| ERR_IRD_STATUS_FAILED |
| ERR_IRC_STATUS_SUCCESS |
| ERR_IRC_STATUS_PENDING |
| ERR_IRC_STATUS_FAILED |
| ERR_OBX_STATUS_SUCCESS |
| ERR_OBX_STATUS_FAILED |
| ERR_OBX_STATUS_PENDING |
| ERR_OBX_STATUS_DISCONNECT |
| ERR_OBX_STATUS_NO_CONNECT |
| ERR_OBX_STATUS_MEDIA_BUSY |

| **Error Messages** |
|---|
| ERR_OBX_STATUS_INVALID_HANDLE |
| ERR_OBX_STATUS_PACKET_TOO_SMALL |
| ERR_OBX_STATUS_BUSY |
| **Networking services error message** |
| ERR_NET_NO_BUFFER |
| ERR_NET_USER |
| ERR_NET_PARAMETER |
| ERR_NET_HOST |
| ERR_NET_HARDWARE |
| ERR_NET_UNKNOWN_DOMAIN_NAME |
| ERR_NET_HANDSHAKE |
| **Text management services error messages** |
| ERR_TXT_ID |
| ERR_TXT_IC_ID |
| ERR_TXT_GC_ID |
| ERR_TXT_MAPPING |
| ERR_TXT_FONT_TYPE |
| ERR_TXT_FONT_STYLE |
| ERR_TXT_FONT_COLOR |
| ERR_TXT_CURSOR_POS |
| ERR_TXT_X_COOR |
| ERR_TXT_Y_COOR |
| ERR_TXT_TMPLT_WT |
| ERR_TXT_TMPLT_HT |
| ERR_TXT_CURSOR_X |
| ERR_TXT_CURSOR_Y |
| ERR_TXT_ZERO_SPC |
| ERR_TXT_FORMAT |
| ERR_TXT_NOT_FIT |
| ERR_TXT_LINE_WT |

| Error Messages |
|---|
| ERR_TXT_FONT_SIZE |
| ERR_TXT_BIT_LEN |
| ERR_TXT_NULL_STRING |
| ERR_TXT_PAN_INIT |
| ERR_TXT_LCD_X |
| ERR_TXT_LCD_Y |
| ERR_TXT_LCD |
| ERR_TXT_GPX |
| ERR_TXT_ARG |
| **Graphic manipulation services error message** |
| ERR_GPX_GC_ID |
| ERR_GPX_NO_MEM |
| ERR_GPX_NUM |
| ERR_GPX_COLOR |
| ERR_GPX_STYLE |
| ERR_GPX_COORDINATE |
| ERR_GPX_X_POS |
| ERR_GPX_Y_POS |
| ERR_GPX_WIDTH |
| ERR_GPX_HEIGHT |
| ERR_GPX_PAN_INIT |
| ERR_GPX_PAN_ADDRESS |
| ERR_GPX_PAN_WIDTH |
| ERR_GPX_PAN_HEIGHT |
| ERR_GPX_LCD_X |
| ERR_GPX_LCD_Y |
| ERR_GPX_LCD_RADIUS |
| ERR_GPX_LCD_FONT |
| ERR_GPX_CURSOR_INIT |
| ERR_GPX_DOT_WIDTH |

| Error Messages |
|---|
| ERR_GPX_FILL_PATTERN |
| ERR_GPX_FILL_SPACE |
| **Software Keyboard module error codes** |
| ERR_SKY_ID |
| ERR_SKY_TASK_ID |
| ERR_SKY_APP_ID |
| ERR_SKY_IC_ID |
| ERR_SKY_GC_ID |
| ERR_SKY_PANNING_SCREEN |
| ERR_SKY_NOT_USE |
| ERR_SKY_USED |
| ERR_SKY_KB_WT |
| ERR_SKY_KB_HT |
| ERR_SKY_OPEN |
| ERR_SKY_CLOSE |
| ERR_SKY_XY |
| ERR_SKY_KEY_SIZE |
| ERR_SKY_REPEAT |
| ERR_SKY_NO_KEY |
| **Input pad handling services error messages** |
| ERR_INP_ID |
| ERR_INP_HWRID |
| ERR_INP_DRAWN |
| ERR_INP_NO_CANDIDATE |
| ERR_INP_COORDINATE |
| **Pen input handling services error messages** |
| ERR_PEN_AREA_ID |
| ERR_PEN_RATE |
| ERR_PEN_EVENT_ID |

# Socket Services Additional Error Handling

In addition, PPSM-GT networking services, has additional error handling that enable developer to get more information when error occurred while using most of the networking functions.

Most of the networking functions return a value of -1 when there is an error. The error code is stored in *errno*, and can also be retrieved using the ***getsockopt()*** function, as shown in "Example to retrive networking error messages".

**Listing B.2 Example to retrive networking error messages**

```
int errcode, errlen;

               .i1 = connect(s, (struct Netsockaddr *)&socka,
               sizeof(socka));

               if (i1 < 0)

               {
                     i1 = errno;
                     if (getsockopt(s, SOL_SOCKET, SO_ERROR,
                         &errcode, &errlen) >= 0)
                     i1 = errcode;
                   printf("connect: error %d\n", i1);
                   /* additional error handling */

               }
```

Here the value of *errno* is saved before calling ***getsockopt(),*** in case this call fails and causes *errno* to be overwritten. The ***getsockopt()*** function should be used when possible in PPSM-GT because *errno* is not reentrant.

If a call to ***socket()*** returns -1, there is no socket number to refer to when trying to retrieve the error code. In this case, the error code must be retrieved from *errno*.

The ***gethostbyname_r()*** functions return a pointer to a host data structure. If these functions fail, then a null pointer is returned.

Table B.2 shows the error code and message for the networking services.

**Table B.2 Socket Services Error Messages**

| Error Messages | Error Code | Description |
|---|---|---|
| NE_PARAM | -10 | user parameter error |
| EHOSTUNREACH | -11 | host not reachable |
| ETIMEDOUT | -12 | timeout |
| ECONNABORTED | -14 | protocol error |
| ENOBUFS | -15 | no buffer space |
| EBADF | -16 | connection block invalid |
| EFAULT | -17 | invalid pointer argument |
| EWOULDBLOCK | -18 | operation would block |
| EMSGSIZE | -19 | message too long |
| ENOPROTOOPT | -20 | Protocol not available |
| EDESTADDRREQ | -50 | Destination address required |
| EPROTOTYPE | -52 | Protocol wrong type for socket |
| EPROTONOSUPPORT | -54 | Protocol not supported |
| ESOCKTNOSUPPORT | -55 | Socket Type not supported |
| EOPNOTSUPP | -56 | Operation not supported on socket |
| EPFNOSUPPORT | -57 | Protocol family not supported |
| EAFNOSUPPORT | -58 | Address family not supported by protocol family |
| EADDRINUSE | -59 | Address already in use |
| EADDRNOTAVAIL | -60 | Can't assign requested address |
| ENETDOWN | -61 | Network is down |
| ENETUNREACH | -62 | Network is unreachable |
| ENETRESET | -63 | Network dropped connection because of reset |
| ECONNRESET | -65 | Connection reset by peer |
| EISCONN | -67 | Socket is already connected |

| Error Messages | Error Code | Description |
|---|---|---|
| ENOTCONN | -68 | Socket is not connected |
| ESHUTDOWN | -69 | Can't send after socket shutdown |
| ECONNREFUSED | -72 | Connection refused |
| EHOSTDOWN | -73 | Host is down |
| EALREADY | -76 | operation already in progress |
| EINPROGRESS | -77 | operation now in progress |

C

# How To Make ROM

Making ROM is normally the final steps in a product development cycle. Application program developed in the RAM development environment, is now really to be tested in the ROM or Flash environment; an environment that is similar to the actual product. In this appendix, the following assumption are made and the scope of discussion will be confined to these assumptions.

## Assumptions:

- The target platform is the on-board flash memory for the DragonBall MC68EZ328 and the MC68VZ328 based system boards.
- Embedded 68K CodeWarrior and SDS singe step compiler and linker are the developement tools used.
- WinBbug is the download program used for downloading program to the target platform.

This appendix consists of the following sections:

## Making ROM Fundamentals

The method of building a program on the target board flash memories involved the followings:

1. A flash writing program that is able to write to the Flash memory of the target board with the block of data in the RAM area of the target board.
2. The blocks of data in the RAM area are actually the application program of the device chopped into blocks of data.
3. A download program that is able to download the flash writing program and the blocks of data into the RAM area.

## Making ROM Procedures

Making the ROM will focus on what need to be done to get the create the flash writing program, and how to use the download utility program supply with PPSM-GT.

### Before making the ROM

Before making the ROM, the following are the preparation works:

#### Find out the memory map of the target board

It is essential to partition the available memory space of the system development board for different uses. Typical mapping includes regions for text, initialized data, un-initialized data and dynamic heap memory (for malloc()).

#### Read the data sheet and write a flash copy program

Different flash memories commonly have different flash-burning procedures. The flash copy program must hence be tailor-made to each flash model in order to write data into that particular flash model properly. An example of a flash copy program for the VZ-ADS board is attached in the Code Examples Listing C.6.

### Make the target program workable for download first

The target program shall be tested on the on-board RAM using CodeWarrior first. Ensuring a workable program before writing to flash could reduce debug time.

### Prepare an initialization B-record

The chip selects of the MC68VZ328 processor on the development board can be initialized with a B-record file. The WinBug utility is used in this step. After the chip selects are properly configured, the memory map of the target broad shall be as expected. An example for VZ-ADS is included in the Code Examples under VZADS.LCF.

### Write an assembly boot code

The boot code shall be able to configure the system clock and chip selects for memory mapping. At the end of the boot code a jump statement is usually included to start the OS or kernel. For PPSM-GT, the boot code shall jump to a function named “START”. The followings are the:

- Boot Strap Code requirement(boot.s)
- 68K Start-up requirement
- Chip Selects requirement
- Peripheral Devices requirement.

#### *Boot Strap Code requirement(boot.s)*

The boot strap code performs the following functions:

- Starts the 68K core upon reset
- Map the chip-selects of MC68VZ328 to run on a particular hardware platform
- Initialization of peripheral devices on the MC68VZ328
- Jump into PPSM-GT start-up code

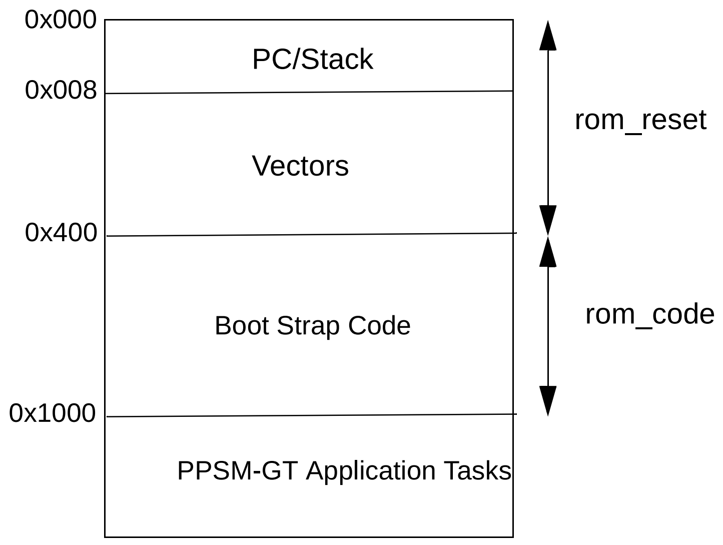
Depending on the size and address of ROM that are used, the chip selects inside boot.s need to be changed accordingly. An example of the boot.s can be found in the Code Examples under boot.s

### *68K Start-up requirement*

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-GT in the linker specification file to perform the mapping.

**Figure C.1 Memory map for boot strap code**



**ROM_RESET** This is used to map the 68K first 256 locations. In the boot strap code, it is defined as:

```
SECTIONrom_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.

**ROM_CODE** This regions is declared to store the boot strap code. Because this code is NOT part of PPSM-GT 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-GT start-up code. The label START is pre-defined as the start location for the startup code.

### *Chip Selects requirement*

For the M68VZ328ADS development board, Chip-Select group A is used for ROM and Chip-Select group B is used for RAM. Please refer to the MC68VZ328 Integrated Processor User’s Manual, MC68328UM/AD, for details on chip select programming.

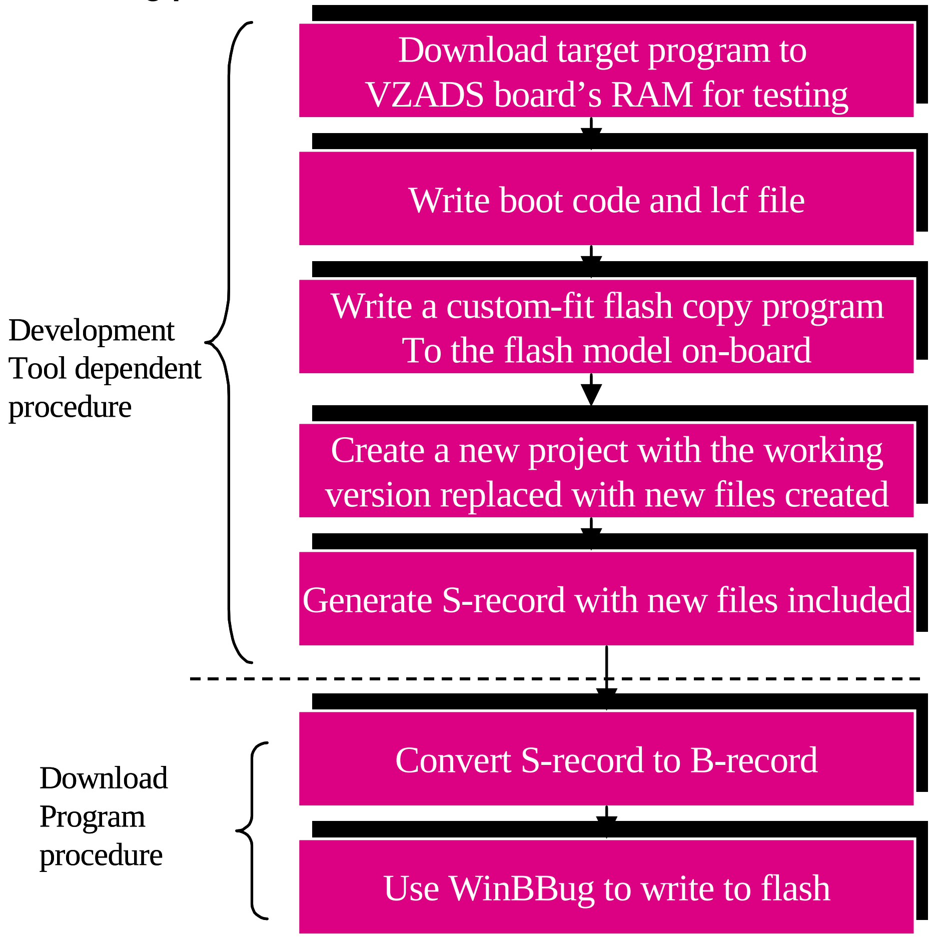
### *Peripheral Devices requirement*

Initialization of the peripherals such as default interrupt vector, watchdog and LCD controller. Please refer to the MC68VZ328 Integrated Processor User’s Manual, MC68VZ328UM/AD, for details on chip select programming.

## Making the ROM

Making the ROM requires two procedures as shown in Figure C.2. There is a development tool dependent procedures and a download program procedure. The development tools dependent procedures is unique and development toold specified. In this appendix, only the Metrowerks code warrior and the SDS single step procedures will be discussed.

**Figure C.2 ROM making procedure**


Download target program to VZADS board's RAM for testing
Write boot code and lcf file
Write a custom-fit flash copy program To the flash model on-board
Create a new project with the working version replaced with new files created
Generate S-record with new files included
Convert S-record to B-record
Use WinBBug to write to flash
Development Tool dependent procedure
Download Program procedure

### Metrowerk Code Warrior procedure

#### *Write a linker command file*

A linker command file (lcf) instructs the CodeWarrior linker where in memory to place each segment of the program. For details on writing a lcf, please refer to the CodeWarrior manual – Targeting Embedded 68K, and Linker Command File Syntax. A detailed example is included in the Code Examples under Listing C.4.

When writing a program for PPSM-GT, attention shall be drawn to the following four values:

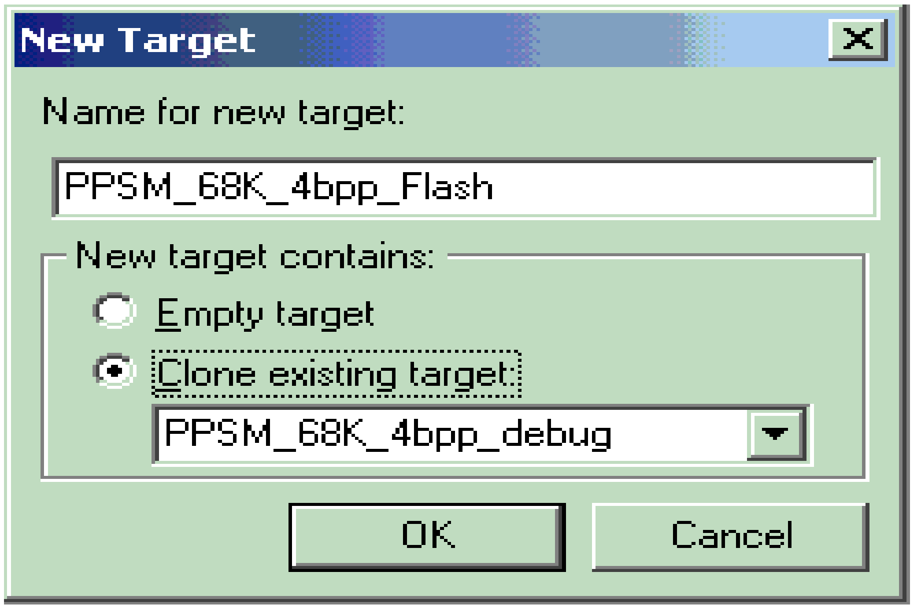
| | |
|---|---|
| _startof_bss | Starting address of initialized data. Have to be aliases with word (16-bit). |
| _sizeof_bss | The size of initialized data in byte. |

| | |
|---|---|
| ___heap_addr | Starting address of heap memory. Have to be aliases with word (16-bit). |
| ___heap_size | The size of heap memory. |

### Use CodeWarrior to output an S-record

1. Create a new project (for flash) by cloning the working, tested project as shown in Figure C.3
2. Add the prepared linker command file (VZADS_ROM.lcf), boot code file (Reset.s), and flash copy program (flash_m68328vz_ads.s) to the newly created projected.
3. De-select the old lcf and select the new lcf so the linker knows which one to read. Select the other two newly added files as well as shown in Figure C.4.
4. Build the new project with "Generate S-record" option enabled.
5. Proceed to "Convert the S-record to B-record" on page 490

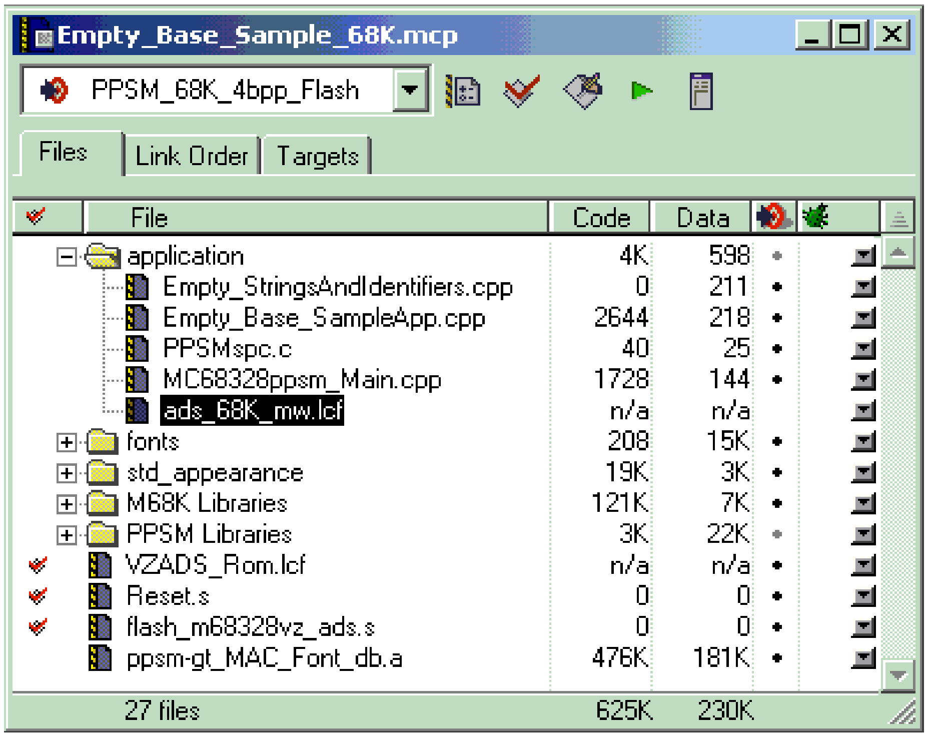
**Figure C.3 Creating a new target**



PPSM_68K_4bpp_Flash

[✓ Clone existing target:] PPSM_68K_4bpp_debug

**Figure C.4 Selecting the new link specified file**



| ✓ | File | Code | Data | | |
|---|---|---|---|---|---|
| | application | 4K | 598 | • | |
| | Empty_StringsAndIdentifiers.cpp | 0 | 211 | • | |
| | Empty_Base_SampleApp.cpp | 2644 | 218 | • | |
| | PPSMspc.c | 40 | 25 | • | |
| | MC68328ppsm_Main.cpp | 1728 | 144 | • | |
| | ads_68K_mw.lcf | n/a | n/a | | |
| | fonts | 208 | 15K | • | |
| | std_appearance | 19K | 3K | • | |
| | M68K Libraries | 121K | 7K | • | |
| | PPSM Libraries | 3K | 22K | • | |
| ✓ | VZADS_Rom.lcf | n/a | n/a | • | |
| ✓ | Reset.s | 0 | 0 | • | |
| ✓ | flash_m68328vz_ads.s | 0 | 0 | • | |
| | ppsm-gt_MAC_Font_db.a | 476K | 181K | • | |
| | 27 files | 625K | 230K | | |

## SDS single step procedure

**NOTE** This section is for SDS single step user only.

## SDS Linker Supplications File for ROM

The SDS Linker Supplications File for ROM as shown in Listing C.1 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.

**Listing C.1 SDS 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

## Generating S-Record File

After the PPSM-GT 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.

### 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 |

### 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> |

**Listing C.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.

**Listing C.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.

## Convert the S-record to B-record

The S-record file is not yet ready to be copied to flash; it must be converted to a B-record file first. The stob.exe utility is a convenient tool for this conversion process. The utility has an intuitive interface hence the steps are not detailed in this document.

### Using Windows Bbug to write the flash memory and run the target program

Steps in operating WinBbug and running the target program:

1. Switch the VZ-ADS board to bootstrap mode.
2. Under the platform menu, select processor model and which UART the PC is connected to.
3. Select Communications (Ctrl+U) and set the PC terminal details.
4. Select Init (Ctrl+I) and choose the prepared initialization B-record.
5. Load the target program into RAM by selecting Load (Ctrl+L). Choose the target program B-record in the dialogue box.
6. Type Go <*address*> to execute the flash copy program where <*address*> is specified in the lcf file (where the flash copy program starts in RAM).
7. Reset the VZ-ADS board and power off. Switch the VZ-ADS out og bootstrap mode then power on.
8. The target program shall start running.

## Summary

This appendix outlines the procedure to build and run a program from the on-board flash memory for the DragonBall MC68EZ328 and the MC68VZ328 based system boards. Readers are assumed to use embedded 68K CodeWarrior as their compiler and linker and are expected to have fully tested their target program in RAM. Finally, the utility WinBbug utility program is used for downloading the program into the ADS board.

**NOTE** The examples throughout this document are based on the M68VZ328 ADS v1.0 (VZ-ADS) with DragonBall-VZ (MC68VZ328) system development board.

## Code Examples

**Listing C.4 VZADS.LCF**

```
/* Sample Linker Command File for M683VZ28ADS ver 1.0 */
/* Location to reconfig if using custom hardware instead of
M68VZ328ADS */

/* 1. RAM buffer holding ROM image before flashing */
/* 2. Some RAM location to hold flash program */
/* 4. beginning of application data RAM, leaving room */
/*    in case we wish to use RAM exception vector later */
/*    (beware of vector table or monitor footprint in RAM) */
                                                          */
MEMORY {
    FLASH_BUFFER(RWX) : ORIGIN = 0x00010000,    LENGTH = 0x00
    FLASH_CODE(RX) :    ORIGIN = 0x8000,        LENGTH = 0x8000
    TEXT (RX) : ORIGIN = 0x01000000,           LENGTH = 0x200000
    DATA (RWX): ORIGIN = 0x00000400,            LENGTH = 0x200000
}

KEEP_SECTION{ .reset}
FORCE_ACTIVE{ copy_to_flash}
```

```
SECTIONS {

    .flash_buffer :
    {
        ___FBUF_START = .;
    } > FLASH_BUFFER

    .flash :
    {
        *(.flashinit)
    } > FLASH_CODE

    /* init boot code that must be run in ROM */
    /* code and read only data in ROM */
    .main_application : AT(ADDR(.flash_buffer))
    {
        ___FLASH_START = .;
        *(.reset)
        . = ALIGN(0x4);
        *(.text)
        . = ALIGN(0x4);
        *(.rodata)
        . = ALIGN(0x4);
    } > TEXT

    /* initialized data and C++ code will be copy to RAM by
runtime function */
    .cpp_code : AT(ADDR(.flash_buffer) +
SIZEOF(.main_application))
    {
        _DATA_ROM = ADDR(.main_application) +
SIZEOF(.main_application);
        _DATA_RAM = .;

        . = ALIGN(0x4);
        *(.exception)

        . = ALIGN(0x4);
        __exception_table_start__ = .;
        EXCEPTION
        __exception_table_end__ = .;
        . = ALIGN(0x4);
        ___sinit__ = .;
        STATICINIT

        . = ALIGN(0x4);
    } > DATA

    .data : AT(ADDR(.flash_buffer) + SIZEOF(.main_application) + 
SIZEOF(.cpp_code))
    {
        . = ALIGN(0x4);
        __START_DATA = .;
        *(.data)
        __END_DATA = .;

        . = ALIGN(0x4);
        __START_SDATA = .;
        *(.sdata)
        __END_SDATA = .;

        . = ALIGN(0x4);
        __SDA_BASE = .;             /* A5 set to  middle of data 
and bss */
    } >> DATA

    /* uninitialized data in RAM */
    .uninitialized_data :
    {
        . = ALIGN(0x4);
        __START_SBSS = .;
        *(.sbss)
        *(SCOMMON)
        __END_SBSS = .;

        . = ALIGN(0x4);
        __START_BSS = .;
        *(.bss)
        *(COMMON)
        __END_BSS = .;

        . = ALIGN(0x4);
        __START_HEAP = .;
    } >> DATA

    /* ROM table is a list record of source, destination and size
*/
    /* for memory locations need to be copy from ROM to RAM, last
*/
    /* record have null in all field */
    .romp : AT(ADDR(.flash_buffer) + SIZEOF(.main_application) +
SIZEOF(.cpp_code) + SIZEOF(.data))
    {
        __S_romp = ADDR(.main_application) +
SIZEOF(.main_application) + SIZEOF(.cpp_code) + SIZEOF(.data);
        WRITEW(_DATA_ROM);
        WRITEW(_DATA_RAM);
        WRITEW(SIZEOF(.cpp_code) + SIZEOF(.data));
        WRITEW(0);
        WRITEW(0);
        WRITEW(0);
    }

    ___FBUF_END = ADDR(.flash_buffer) + SIZEOF(.main_application)
+ SIZEOF(.cpp_code) + SIZEOF(.data) + SIZEOF(.romp);

    /* The following values must be defined for PPSM-GT */
    ___heap_addr = __START_HEAP;    /* heap grows in opposite
direction of stack */
    ___heap_size = 0x50000;         /* heap size set to 0x50000
bytes (500KB) */

    _startof_bss = __START_BSS;
    _sizeof_bss = __END_BSS - __START_BSS;
}
```

**Listing C.5 boot.s**

```
;*****************************************************************
**;
;   This is the boot routine for the MC68VZ328 ADS board. Product
; engineers should examine all the configurations carefully and
; change them according to their system requirements.
;*****************************************************************
**;

MON_BOOT.equu___reset; Boot entry point
MON_STACKTOP.equ$4100; Initial stack

M328BASE  .equ $FFFFF000;  Base address for system registers

; SIM28 System Configuration Registers

SCR   .equ(M328BASE+$000)

; Chip Select Registers
; CS Group Base Registers
GRPBASEA.equ(M328BASE+$100)
GRPBASEB.equ(M328BASE+$102)
GRPBASEC.equ(M328BASE+$104)
GRPBASED.equ(M328BASE+$106)
; CS Registers
CSA   .equ(M328BASE+$110)
CSB   .equ(M328BASE+$112)
CSC   .equ(M328BASE+$114)
CSD   .equ(M328BASE+$116)

CSCR  .equ(M328BASE+$10A)
DRAMCFG.equ(M328BASE+$C00)
DRAMMC  .equ(M328BASE+$C00)
DRAMCTL.equ(M328BASE+$C02)
DRAMC   .equ(M328BASE+$C02)
SDCTRL  .equ(M328BASE+$C04)
EMUCS .equ(M328BASE+$118)
CSCTR .equ(M328BASE+$150)

; PLL Registers

PLLCR .equ(M328BASE+$200); Control Reg
PLLFSR  .equ(M328BASE+$202); Freq Select Reg
PLLTSR  .equ(M328BASE+$204); Test Reg

; Power Control Registers
```

```
PCTLR .equ(M328BASE+$206); Control Reg

; Interrupt Registers

IVR   .equ(M328BASE+$300); Interrupt Vector Reg
ICR   .equ(M328BASE+$302); Interrupt Control Reg
IMR   .equ(M328BASE+$304); Interrupt Mask Reg
ISR   .equ(M328BASE+$30C); Interrupt Status Reg
IPR   .equ(M328BASE+$310); Interrupt Pending Reg

; PIO Registers

; Port A Registers
PADIR .equ(M328BASE+$400); Direction Reg
PADATA  .equ(M328BASE+$401); Data Reg
PAPUEN  .equ(M328BASE+$402); Pullup Enable Reg

; Port B Registers
PBDIR .equ(M328BASE+$408); Direction Reg
PBDATA  .equ(M328BASE+$409); Data Reg
PBPUEN  .equ(M328BASE+$40A); Pullup Enable Reg
PBSEL .equ(M328BASE+$40B); Select Reg

; Port C Registers
PCDIR .equ(M328BASE+$410); Direction Reg
PCDATA  .equ(M328BASE+$411); Data Reg
PCPUEN  .equ(M328BASE+$412); Pullup Enable Reg
PCPDEN  .equ(M328BASE+$412); Pull-down Enable Reg
PCSEL .equ(M328BASE+$413); Select Reg

; Port D Registers
PDDIR .equ(M328BASE+$418); Direction Reg
PDDATA  .equ(M328BASE+$419); Data Reg
PDPUEN  .equ(M328BASE+$41A); Pullup Enable Reg
PDSEL .equ(M328BASE+$41B); Select Reg
PDPOL .equ(M328BASE+$41C); Polarity Reg
PDIRQEN .equ(M328BASE+$41D); IRQ Enable Reg
PDIRQEDGE.equ(M328BASE+$41F); IRQ Edge Reg

; Port E Registers
PEDIR .equ(M328BASE+$420); Direction Reg
```

```
PEDATA  .equ(M328BASE+$421); Data Reg
PEPUEN  .equ(M328BASE+$422); Pullup Enable Reg
PESEL .equ(M328BASE+$423); Select Reg

; Port F Registers
PFDIR .equ(M328BASE+$428); Direction Reg
PFDATA  .equ(M328BASE+$429); Data Reg
PFPUEN  .equ(M328BASE+$42A); Pullup Enable Reg
PFSEL .equ(M328BASE+$42B); Select Reg

; Port G Registers
PGDIR .equ(M328BASE+$430); Direction Reg
PGDATA  .equ(M328BASE+$431); Data Reg
PGPUEN  .equ(M328BASE+$432); Pullup Enable Reg
PGSEL .equ(M328BASE+$433); Select Reg

PKSEL .equ(M328BASE+$443); Select Reg
PMSEL .equ(M328BASE+$44B); Select Reg

; PWM Registers
PWMC  .equ(M328BASE+$500); Control Reg
PWMS  .equ(M328BASE+$502); Sample Reg
PWMCNT  .equ(M328BASE+$504); Counter

; Timer Registers
; Timer 1 Registers
TCTL1 .equ(M328BASE+$600); Control Reg
TPRER1  .equ(M328BASE+$602); Prescalar Reg
TCMP1 .equ(M328BASE+$604); Compare Reg
TCR1  .equ(M328BASE+$606); Capture Reg
TCN1  .equ(M328BASE+$608); Counter
TSTAT1.equ(M328BASE+$60A); Status Reg

; Watchdog Registers
WCR   .equ(M328BASE+$B0A); Control Reg

; SPI Registers
; SPI Master Registers
SPIMDATA.equ(M328BASE+$800); Control/Status Reg
SPIMCONT.equ(M328BASE+$802); Data Reg

; UART Registers
```

```
USTCNT  .equ(M328BASE+$900); Status Control Reg
UBAUD   .equ(M328BASE+$902); Baud Control Reg
UARTRX  .equ(M328BASE+$904); Rx Reg
UARTTX  .equ(M328BASE+$906); Tx Reg
UARTMISC.equ(M328BASE+$908); Misc Reg
UARTNIPR.equ(M328BASE+$90A) ; Non-Integer Prescalar Reg

; LCDC Registers
LSSA  .equ(M328BASE+$A00); Screen Start Addr Reg
LVPW  .equ(M328BASE+$A05); Virtual Page Width Reg
LXMAX .equ(M328BASE+$A08); Screen Width Reg
LYMAX   .equ(M328BASE+$A0A); Screen Height Reg
LCXP  .equ(M328BASE+$A18); Cursor X Position
LCYP  .equ(M328BASE+$A1A); Cursor Y Position
LCWCH   .equ(M328BASE+$A1C); Cursor Width & Height Reg
LBLKC .equ(M328BASE+$A1F); Blink Control Reg
LPICF .equ(M328BASE+$A20); Panel Interface Config Reg
LPOLCF  .equ(M328BASE+$A21); Polarity Config Reg
LACDRC  .equ(M328BASE+$A23); ACD (M) Rate Control Reg
LPXCD .equ(M328BASE+$A25); Pixel Clock Divider Reg
LCKCON  .equ(M328BASE+$A27); Clocking Control Reg
LRRA  .equ(M328BASE+$A29); Last Buffer Addr Reg
LOTCR .equ(M328BASE+$A2B); Octet Terminal Count Reg
LPOSR .equ(M328BASE+$A2D); Panning Offset Reg
LFRCM .equ(M328BASE+$A31); Frame Rate Control Mod Reg
LGPMR .equ(M328BASE+$A32); Gray Palette Mapping Reg
LIRQR .equ(M328BASE+$A34); Interrupt Control Reg

; RTC Registers
RTCHMSR.equ(M328BASE+$B00); Hrs Mins Secs Reg
RTCALM0R.equ(M328BASE+$B04); Alarm Register 0
RTCDAY  .equ(M328BASE+$B08); RTC date reg
RTCWD .equ(M328BASE+$B0A); RTC watch dog timer reg
RTCCTL  .equ(M328BASE+$B0C); Control Reg
RTCISR.equ(M328BASE+$B0E); Interrupt Status Reg
RTCIENR.equ(M328BASE+$B10); Interrupt Enable Reg
RSTPWCH   .equ(M328BASE+$B12) ; Stopwatch Minutes

; ICEM registers
ICEMACR   .equ (M328BASE+$D00)
ICEMAMR   .equ (M328BASE+$D04)
ICEMCCR   .equ (M328BASE+$D08)
```

```
ICEMCMR    .equ (M328BASE+$D0A)
ICEMCR    .equ (M328BASE+$D0C)
ICEMSR    .equ (M328BASE+$D0E)

*****************************************************************
*************
*       RESET OPTIONS
*****************************************************************
*************

    .section .reset
rom_base:
;-- SECTIONrom_reset - SP, start addr & space for Exception
Vectors

    .DC.LMON_STACKTOP; stack pointer
    .DC.LMON_BOOT; program counter
    .skip(62*4); space for Motorola defined Exception Vectors
    .skip(192*4); space for the 192 User defined Exception
Vectors

    .global ___reset
___reset:

    ;************************************************
    ;* System initialization                        *
    ;************************************************
    move.b  #$18,SCR            ; Disable Double Map

    ;************************************************
    ;* Primary boot image is at start of flash.
    ;* Secondary boot image is at start+0x10000.
    ;* If this is the primary image and PD2 is low,
    ;* boot alternate image.
    ;************************************************
    lea.l0(PC), A0; get PC
    move.lA0, D0
    and.l#$10000, D0; is this secondary image?
    bne.sboot_trk; if so, don’t check switch,
              ; just boot this image

    ori.b   #$0F,PDSEL
    move.b  #$03,PDDIR
    move.b  #$FF,PDPUEN
    move.b  PDDATA,D0
    andi.b  #$04,D0
    bne.sboot_trk; if PD2 high, boot this image

    move.l$01010000, SP; otherwise boot alternate image
    move.l$01010004, A0
    jmp (A0)

    ;*************************************************
    ; Booting MetroTRK
    ;*************************************************
boot_trk:
    move.b  #$9,PGSEL            ; config PG0/DTACK to GPI/O,input
    ;move.w  #$2480,PLLCR         ; ??MHz Sysclk, enable clko
    move.w  #$2400,PLLCR         ; ??MHz Sysclk, enable clko

    move.l  #MON_STACKTOP,A7     ; Install stack pointer
    move.w  #$2700,sr            ; mask off all interrupts
    move.w  #$00,RTCWD           ; disable watch dog
    move.w  #$08,ICEMCR          ; disable ICEM vector hardmap
    move.w  #$07,ICEMSR          ; clear level 7 interrupt

    ;*******************************
    ;* Port Initialization        *
    ;*******************************
    move.b  #$03,PFSEL           ; select A23-A20, CLKO, CSA1
    move.b  #$00,PBSEL           ; Config port B for chip select
A,B,C and D
    move.b  #$00,PESEL           ; select *DWE
    move.b  #$F1,PKSEL
    move.b  #$00,PMSEL


    ;*******************************
    ;* Chip Select initialization *
    ;*******************************
    ;*********************
    ; Flash
    ;*********************
    move.w  #$0800,GRPBASEA     ; GROUPA BASE(FLASH), Start
add.=0x1000000
    move.w  #$0199,CSA          ;

    ;***************************************
    ; SDRAM 64M-bit, Single Band, Latency 2
    ;***************************************
    move.w  #$0000,GRPBASED
    move.w  #$0281,CSD
    move.w  #$0040,CSCR         ; Chip Sel Control Reg

    move.w  #$0000,DRAMC        ; Disable DRAM Controller
    move.w  #$C03F,SDCTRL
    move.w  #$4020,DRAMMC
    move.w  #$8000,DRAMC

    clr.w   d0
delay
    addi.w  #1,d0
    cmp.w   #$FFFF,d0
    bne     delay

    move.w  #$C83F,SDCTRL       ; issue precharge comm
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    move.w  #$D03F,SDCTRL       ; enable refresh
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
```

```
nop
nop
move.w  #$D43F,SDCTRL       ; issue mode command
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop


;************************************************
;* End of System initialization                 *
;************************************************

clr.l       d0
clr.l       d1
clr.l       d2
clr.l       d3
clr.l       d4
clr.l       d5
clr.l       d6
clr.l       d7

;**********************************************
;* LCD Initization Code                       *
;**********************************************

move.b   #0,PCSEL
move.b   #0,PCPDEN

move.l   #$100403E,LSSA
move.w   #160,LXMAX
move.w   #239,LYMAX
move.b   #10,LVPW
move.b   #$08,LPICF
move.b   #$01,LPOLCF
move.b   #$00,LACDRC
```

```
    move.b   #$02,LPXCD
    move.b   #$14,LRRA
    move.b   #$00,LPOSR
    move.b   #$00,LCKCON           ; disable LCDC
    move.b   #$80,LCKCON           ; enable LCDC, 0ws, 16-bit


;*****************************************************************
******
;
; Program Interrupt Controller
;
;*****************************************************************
******

    move.b #$40,IVR
    move.l #$007FFFFF,IMR           ;enable NMI interrupt


;*****************************************************************
******
;
; Runtime initialization
;
;*****************************************************************
******

    .extern START
    JMP START; jump to MW startup code
```

**Listing C.6 flash_m68328vz_ads.s**

```
;
; Code to copy data from memory into Fujitsu MBM29LV160T flash.
; It assumes 1 flash chip in word mode.
; This code assumes a top-boot device.  It also assumes
; that the starting flash address is at the beginning
; of a sector.
;

FLASH_BASE.equ$01000000

CPU_SPEED.equ16 ; 16 MHz
DLOOP_CYCLES.equ26; SUBI.L #,Dn -> 16; BNE.S -> 10
DELAY_TIME.equ20; 20ms max delay after sector write
DELAY_COUNT.equ(CPU_SPEED*1000000*DELAY_TIME/1000)/DLOOP_CYCLES

    .section .flashinit
    .extern ___FBUF_START
    .extern ___FBUF_END
    .extern ___FLASH_START
    .globalcopy_to_flash
copy_to_flash:

    ;
    ; Set up registers:
    ; a0 - flash image source start addr (in RAM)
    ; a1 - flash image source end addr (in RAM)
    ; a2 - flash image dest addr (in FLASH)
    ;

    move.l#___FBUF_START,a0; source addr of flash image
    move.l#___FBUF_END,a1; end addr of source flash image
    move.l#___FLASH_START,a2; dest addr of flash image

    move.l#$555*2,d1; load command offset 1 to d1
    move.l#$2aa*2,d2; load command offset 2 to d2


    ;
    ; Erase the next sector.  Each sector must be erased before it
    ; can be programmed.
    ;

erase_loop:
    cmp.la0,a1  ; if done copying, verify
    ble do_compare

    move.la2,d0
    and.l#$ffe00000,d0; calc. base addr of current chip
    move.ld0,a3

    move.w#$aa,(d1.l,a3); unlock step 1
    move.w#$55,(d2.l,a3); unlock step 2
    move.w#$80,(d1.l,a3); sector erase setup
    move.w#$aa,(d1.l,a3); unlock step 1
    move.w#$55,(d2.l,a3); unlock step 2
    move.w#$30,(a2); erase current sector
erase_verify_loop:
    move.w(a2),d0; check sector data
    cmp.w#$ffff,d0; erased?
    bne.serase_verify_loop; if not, keep checking


do_program:
    ; Get the sector size, which depends on the sector offset.
    ; This code assumes a top-boot device.  It also assumes
    ; that the starting flash address is at the beginning
    ; of a sector.

    move.la2,d0
    sub.la3,d0  ; get offset of sector
    cmp.l#$001f0000,d0; is it sa31 or higher?
    bge.scheck_sa31; if so, do more tests
    move.l#$00010000,d0; otherwise, size is 64K
    bra program_loop

check_sa31:
    cmp.l#$001f8000,d0; is it sa32 or higher?
    bge.scheck_sa32; if so, do more test
    move.l#$00008000,d0; otherwise, size is 32K
    bra program_loop

check_sa32:
    cmp.l#$001fc000,d0; is it sa34?
    bge.ssize_sa34; if so, get sa34 size
    move.l#$00002000,d0; otherwise, size is 8K
    bra program_loop

size_sa34:
    move.l#$00004000,d0; sa34 size is 16K
```

```
program_loop:

    move.w#$aa,(d1.l,a3); unlock step 1
    move.w#$55,(d2.l,a3); unlock step 2
    move.w#$a0,(d1.l,a3); program command

    move.w(a0),d3
    move.wd3,(a2); write data to flash
program_verify_loop:
    cmp.w(a2),d3; data written?
    bne.sprogram_verify_loop; if not, wait

    add.l#2,a0  ; next word
    add.l#2,a2  ; next word

    cmp.la0,a1  ; done copying?
    ble.sdo_compare; if so, verify

    sub.l#2,d0  ; next word
    beq erase_loop; if end of sector, erase next
    bra.sprogram_loop; otherwise, copy next word


    ; Verify that the flash contents were written correctly.
do_compare:
    move.l#___FBUF_START,a0; source addr of flash image
    move.l#___FBUF_END,a1; end addr of source flash image
    move.l#___FLASH_START,a2; dest addr of flash image

compare_loop:
    cmp.w(a0)+,(a2)+
    bne prog_fail

    cmp.la0,a1  ; is entire image verified?
    bgt compare_loop; if not, repeat

    trap#0      ; done
    nop

prog_fail:
    trap#1      ; failed
    nop
```

**Listing C.7 Initialization Code**

```
********************************************************
* VZ-ADS Init B-Record
*
********************************************************
FFFFF0000118  SCR init Disable Double Map
FFFFFB0B0100  Disable WD
FFFFF42B0103  enable clko
FFFFF40B0100  enable chip select
FFFFFD0D0108  disable hardmap
FFFFFD0E0107  clear level 7 interrupt
FFFFF4230100  set PE3 as *DWE

FFFFF3000140        IVR
FFFFF30404007FFFFF  IMR

***
CSA
***
FFFFF100020800 Group Base Add 16M
FFFFF110020199 Chip Sel

************
SDRAM Config
************
FFFFF44301F1 PKSEL
FFFFF44B0100 PMSEL

CSD
FFFFF106020000 Group Base Add
FFFFF116020281 Chip Sel
FFFFF10A020040 Chip Sel Control

DRAM Contorller
FFFFFC02020000 DRAMC
FFFFFC0402C03F SDRAM Control
```

```
FFFFFC00024020 DRAMMC
FFFFFC02028000 DRAMC

SDRAM Control
FFFFFC0402C83F issue precharge command
FFFFFC0402D03F enable refresh
FFFFFC0402D43F issue mode command

*********
Init LCDC
*********
FFFFF4130100        Disable Port C
FFFFFA00040000403E  LSSA=0x403E
FFFFFA05010A        LVPW
FFFFFA080200A0      LXMAX
FFFFFA0A0200EF      LYMAX
FFFFFA200108        LPICF
FFFFFA210101        LPOLCF
FFFFFA230100        LACD
FFFFFA250102        LPXCD
FFFFFA290114        LRRA
FFFFFA2B0118        LOTCR
FFFFFA2D0100        LPOSR
FFFFFA270100        Disable LCD
FFFFFA270182        Enable LCD
```

D

# PPSM-GT APIs Reference Card

**PPSM-GT API Look Up Card**

| Function Declaration |
|---|
| **Alarm Services** |
| STATUS **AlmCreate**(P_U32 alarmId, U16 year, U8 month, U8 day, U8 hour, U8 minute, U8 type); |
| void **AlmDelete**(ALARM_ID alarmId, TASK_ID taskId); |
| void **AlmDelBefore**(U16 year, U8 month, U8 day, U8 hour, U8 minute, TASK_ID taskId); |
| void **AlmDelAfter**(U16 year, U8 month, U8 day, U8 hour, U8 minute, TASK_ID taskId); |
| void **AlmDeleteAll**(TASK_ID taskId); |
| STATUS **AlmGetCurrent**(ALARM_ID alarmId, P_U32 taskId, P_U16 year, P_U8 month, P_U8 day, P_U8 hour, P_U8 minute, P_U8 type); |
| ALARM_ID **AlmGetId**(void); |
| STATUS **AlmGetIdByTime**(P_U32 alarmId, U16 year, U8 month, U8 day, U8 hour, U8 minute); |
| STATUS **AlmGetNext**(ALARM_ID alarmId, P_U32 taskId, P_U16 year, P_U8 month, P_U8 day, P_U8 hour, P_U8 minute, P_U8 type); |
| STATUS **AlmSetPeriodId**(P_U32 alarmId, U8 period); |
| |
| **Application Services** |
| STATUS **AppAddIC**(APP_ID appId, IC_ID icId); |
| STATUS **AppBindPanInfo**(APP_ID appId, SCREEN_ID panInfo); |

| Function Declaration |
|---|
| STATUS **AppCreate**(P_APP_ID pAppId, P_TEXT pName, P_VOID entryCallback, P_VOID exitCallback, U16 iconWidth, U16 iconHeight, P_U8 pIconImage, U16 ringBufferSize); |
| STATUS **AppCreateGC**(P_GC_ID pGCId, U16 horz, U16 vert); |
| STATUS **AppCreatePanScreen**(P_SCREEN_ID pScreenId, U16 horz, U16 vert); |
| STATUS **AppDelete**(APP_ID appId); |
| STATUS **AppDeleteGC**(GC_ID gcId); |
| STATUS **AppDeletePanScreen**(SCREEN_ID screenId); |
| STATUS **AppGetAppIdFromIC**(IC_ID icId, P_APP_ID pAppId); |
| APP_ID **AppGetCurrent**(); |
| STATUS **AppGetCurrPanScreen**(P_SCREEN_ID pScreenId); |
| STATUS **AppGetDownAppId**(P_EVENT pEvent, P_APP_ID pAppId); |
| STATUS **AppGetFirstIC**(APP_ID appId, P_IC_ID pIC); |
| STATUS **AppGetIcon**(APP_ID appId, P_U16 pIconWidth, P_U16 pIconHeight, P_U8 * pIconImage); |
| STATUS **AppGetName**(APP_ID appId, P_TEXT * pName); |
| STATUS **AppGetNext**(APP_ID appId, P_APP_ID pAppId); |
| STATUS **AppGetPanInfo**(APP_ID appId, SCREEN_ID * pPanInfo); |
| STATUS **AppGetPanScreen**(GC_ID gcId, P_SCREEN_ID pScreenId); |
| U16 **AppGetPanScreenHeight**(void); |
| U16 **AppGetPanScreenWidth**(void); |
| STATUS **AppGetPrev**(APP_ID appId, P_APP_ID pAppId); |
| STATUS **AppMoveICToTop**(APP_ID appId, IC_ID icId); |
| STATUS **AppRemoveIC**(IC_ID icId); |
| STATUS **AppRemovePanInfo**(APP_ID appId); |
| STATUS **AppSetCurrPanScreen**(SCREEN_ID screenId); |
| STATUS **AppSetPanScreen**(GC_ID gcId, SCREEN_ID screenId); |
| STATUS **AppSwapICList**(APP_ID appId, IC_ID newIC, P_IC_ID pIC); |
| STATUS **AppSwitch**(APP_ID appId); |
| |
| **Audio Handling Services** |

| Function Declaration |
|---|
| STATUS **AudGetCount**(P_U16 playcounter); |
| U16 **AudGetNoteLength**(void); |
| STATUS **AudGetName**(P_U16 pmfname); |
| STATUS **AudGetNumofNote**(P_U16 sumofnote); |
| U8 **AudGetStatus**(void); |
| U16 **AudGetToneDur**(void); |
| STATUS **AudPlayTone**(P_U16 toneData, U32 toneSize, U16 toneDuration, U8 autoRepeat); |
| STATUS **AudPlayMelody**(void); |
| STATUS **AudPauseMelody**(void); |
| STATUS **AudPlayWave**(P_U8 waveData, U32 waveSize, U8 samplingRate); |
| STATUS **AudAdvPlayWave**(P_U8 waveData, U32 waveSize, U8 prescaler, U8 repeat, U8 clksel); |
| STATUS **AudSetMelody**(P_PMF pmf); |
| STATUS **AudSetTone**(void); |
| STATUS **AudStopMelody**(void); |
| STATUS **AudStopTone**(void); |
| STATUS **AudStopWave**(void); |
| |
| **Download Application Services** |
| STATUS **AppCreateTask**(P_TASK_ID pTaskId, APP_ID appId, P_VOID pFunc, U32 stackSize); |
| STATUS **AppConvertImage**(U32 imagePtr, P_APP_ID pAppId, P_TASK_ID pTaskId); |
| STATUS **AppDeleteImage**(APP_ID appId); |
| STATUS **AppDeleteTask**(APP_ID appId, TASK_ID taskId); |
| |
| **Event Management Services** |
| STATUS **EvtAddToChannel**(TASK_ID taskId, CHANNEL_ID channelId); |
| EVTTYPE **EvtAllocType**(VOID) |
| EVTTYPE **EvtCheck**(VOID); |
| BRDCST_ID **EvtCreateBroadcast**(VOID); |

| Function Declaration |
|---|
| CHANNEL_ID **EvtCreateChannel**(VOID); |
| EVTPORT_ID **EvtCreatePort**(VOID); |
| STATUS **EvtDeleteBroadcast**(BRDCST_ID broadcastId); |
| STATUS **EvtDeleteChanne**l(CHANNEL_ID channelId); |
| STATUS **EvtDeletePort**(EVTPORT_ID eventportId); |
| STATUS **EvtDelInQueue**(EVTTYPE) |
| STATUS **EvtFlushQueue**(VOID) |
| STATUS **EvtFreeType**(EVTTYPE) |
| EVTTYPE **EvtGet**(VOID); |
| CHANNEL_ID **EvtGetBrdcstChannel**(BRDCST_ID broadcastId); |
| P_EVENT **EvtGetBrdcstEvent**(BRDCST_ID broadcastId); |
| TASK_ID **EvtGetChLastTask**(CHANNEL_ID channelId); |
| U16 **EvtGetChNumTask**(CHANNEL_ID channelId); |
| TASK_ID **EvtGetChTask**(CHANNEL_ID channelId, P_U32 pTemp); |
| P_EVENT **EvtGetEvent**(VOID); |
| EVTTYPE **EvtGetType**(VOID); |
| U16 **EvtGetUsage**(P_EVENT pEvent); |
| STATUS **EvtInitEvent**(P_EVENT pEvent, EVTTYPE eventType); |
| U8 **EvtIsBrdcstId**(BRDCST_ID broadcastId); |
| U8 **EvtIsChannelId**(CHANNEL_ID channelId); |
| U8 **EvtIsErasable**(P_EVENT pEvent); |
| BOOL **EvtIsTypeAvailable**(EVTTYPE) |
| U8 **EvtIsWakeup**(P_EVENT pEvent); |
| STATUS **EvtRegisterType**(EVTTYPE) |
| STATUS **EvtRmBrdcstFromCh**(BRDCST_ID broadcastId); |
| VOID **EvtRmCurrEvent**(VOID); |
| STATUS **EvtRmTaskFromCh**(TASK_ID taskId, CHANNEL_ID channelId); |
| STATUS **EvtSend**(P_EVENT pEvent, TASK_ID taskId); |
| STATUS **EvtSendUrgent**(P_EVENT pEvent, TASK_ID taskId) |

| Function Declaration |
|---|
| STATUS **EvtSendBrdcstEvent**(BRDCST_ID broadcastId, CHANNEL_ID chaneelId, TICK milliseconds); |
| STATUS **EvtSetBrdcstEvent**(BRDCST_ID broadcastId, P_EVENT pEvent); |
| STATUS **EvtSetErasable**(P_EVENT pEvent); |
| STATUS **EvtSetUnerasable**(P_EVENT pEvent); |
| STATUS **EvtSetUnwakeup**(P_EVENT pEvent); |
| STATUS **EvtSetWakeup**(P_EVENT pEvent); |
| STATUS **EvtWait**(EVTTYPE evtType, TICK milliseconds) |
| STATUS **EvtWaitMultiple**( U16 numOfType, EVTTYPE* pEvtType, TICK milliseconds, BOOL waitAll) |

| Graphic Manipulation Services |
|---|
| STATUS **GpxChangeDisplay**(U16 xPos, U16 yPos); |
| STATUS **GpxDeleteCursor**(SCREEN_ID screenId); |
| STATUS **GpxDrawArc**(U16 x1, U16 y1, U16 x2, U16 y2); |
| STATUS **GpxDrawCircle**(U16 xCenter, U16 yCenter, U16 radius); |
| STATUS **GpxDrawDot**(U16 xPos, U16 yPos); |
| STATUS **GpxDrawEllipse**(U16 xCenter, U16 yCenter, U16 xLength, U16 yLength); |
| STATUS **GpxDrawLine**(U16 xSrc, U16 ySrc, U16 xDest, U16 yDest, U16 dotLine); |
| STATUS **GpxDrawRec**(U16 xSrc, U16 ySrc, U16 xDest, U16 yDest, U16 dotLine); |
| STATUS **GpxDrawVector**(U16 numberOfPoints, P_POINT pPoints, U8 mode); |
| STATUS **GpxExchangeRec**(P_U8 pBitmap, U16 xSrc, U16 ySrc, U16 xDest, U16 yDest); |
| STATUS **GpxFillRec**(U16 xSrc, U16 ySrc, U16 xDest, U16 yDest); |
| STATUS **GpxFillScreen**(COLOR color); |
| U8 **GpxGetBrightness**(VOID); |
| STATUS **GpxGetColor**(P_COLOR pColor); |
| STATUS **GpxGetContrast**(P_DENSITY pLevel1, P_DENSITY pLevel2); |
| STATUS **GpxGetCursorPos**(SCREEN_ID screenId, P_U16 pXPos, P_U16 pYPos); |
| STATUS **GpxGetCursorStatus**(SCREEN_ID screenId, P_U8 pStatus); |
| U16 **GpxGetDisplayHeight**(void); |

| Function Declaration |
|---|
| U16 **GpxGetDisplayWidth**(void); |
| STATUS **GpxGetDisplayOrigin**(SCREEN_ID screenId, P_U16 pXPos, P_U16 pYPos); |
| STATUS **GpxGetDotWidth**(P_U8 pWidth); |
| U8 **GpxGetLCDRefreshRate**(void); |
| STATUS **GpxGetPatternFill**(P_U8 pMode, P_COLOR pBackColor, P_U8 pBorderMode, P_U8 pFillSpace); |
| STATUS **GpxGetStyle**(P_STYLE pStyle); |
| STATUS **GpxInitCursor**(SCREEN_ID screenId); |
| STATUS **GpxInvRec**(U16 xSrc, U16 ySrc, U16 xDest, U16 yDest); |
| STATUS **GpxPutChar**(P_U8 pChar, U16 xPos, U16 yPos, U16 font, U16 width, U16 height); |
| STATUS **GpxPutRec**(P_U8 pBitmap, U16 xSrc, U16 ySrc, U16 xDest, U16 yDest); |
| STATUS **GpxSaveRec**(P_U8 pBitmap, U16 xSrc, U16 ySrc, U16 xDest, U16 yDest); |
| VOID **GpxSetBrightness**(U8 brightness); |
| STATUS **GpxSetColor**(COLOR color); |
| STATUS **GpxSetContrast**(DENSITY level1, DENSITY level2); |
| STATUS **GpxSetCursorBlink**(SCREEN_ID screenId, U8 frequency); |
| STATUS **GpxSetCursorPos**(SCREEN_ID screenId, U16 xPos, U16 yPos); |
| STATUS **GpxSetCursorSize**(SCREEN_ID screenId, U8 cursorWidth, U8 cursorHeight); |
| STATUS **GpxSetCursorStatus**(SCREEN_ID screenId, U8 status); |
| STATUS **GpxSetDisplayOrigin**(SCREEN_ID screenId, U16 xPos, U16 yPos); |
| STATUS **GpxSetDotWidth**(U8 width); |
| VOID **GpxSetLCDRefreshRate**(U8 refreshRateSet); |
| STATUS **GpxSetPatternFill**(U8 mode, COLOR backColor, U8 borderMode, U8 fillSpace); |
| STATUS **GpxSetStyle**(STYLE style); |
| |
| **Handwriting Recognition Handling Services** |
| STATUS **InpClose**(INP_ID id) |
| STATUS **InpDrawPad**(INP_ID inpId) |

| Function Declaration |
|---|
| STATUS **InpGetCandidates**(P_TEXT* pCandidates, P_U16 pNum, P_INP_ID pInpId) |
| STATUS **InpInstallHWR**(P_HWR_ID pId, P_VOID resetEng, P_VOID initEng, P_VOID processStk, P_VOID recgzInp, U32 stackSize) |
| U8 **InpIsInpId(**INP_ID id) |
| U8 **InpIsHWRId**(HWR_ID id) |
| STATUS **InpOpen**(P_INP_ID pId, HWR_ID hwrId, TASK_ID taskId, U16 xPos, U16 yPos, U16 numRow, U16 numCol, U16 areaWidth, U16 areaHeight, TICK timeout, U8 echoSize) |
| STATUS **InpSetColor**(INP_ID id, COLOR color) |
| STATUS **InpSetSamplingRate**(INP_ID id, PEN_RATE time) |
| STATUS **InpTop**(INP_ID id) |
| STATUS **InpUninstallHWR**(HWR_ID id) |

| **Interrupt Service Routine Services** |
|---|
| U8 **IsrGetIrptLv**(U32 module) |
| U8 **IsrIsInUse**(U32 module) |
| STATUS **IsrRelease**(U32 irptFlag); |
| STATUS **IsrRequest**(U32 module, PFIRTHANDLER pfIrptHandler, U32 arg); |
| STATUS **IsrSetIrptLv**(U32 module, U8 irptLevel) |
| |
| **Kernel Services** |
| STATUS **KnlBindGC**(TASK_ID taskId, GC_ID gcId) |
| STATUS **KnlChangePriority**(TASK_ID taskId, U8 priority) |
| SEMA_ID **KnlCreateSemaphore(**VOID) |
| STATUS **KnlCreateTask**(P_TASK_ID pTaskId, P_VOID pFunc, const TEXT pName[], U32 stackSize, S8 priority, KNL_MODE mode) |
| STATUS **KnlCreateTaskWith**(P_TASK_ID pTaskId, P_VOID pFunc, const TEXT pName[], U32 stackSize, U32 arg, S8 priority, KNL_MODE mode) |
| STATUS **KnlDeleteTask**(TASK_ID taskId) |
| STATUS **KnlDelSemaphore**(SEMA_ID semaId, U8 flag) |

| Function Declaration |
|---|
| EVTPORT_ID **KnlGetEventPort**(TASK_ID taskId) |
| STATUS **KnlDisableSwap**(VOID) |
| STATUS **KnlEnableSwap**(VOID) |
| GC_ID **KnlGetGC**(TASK_ID taskId) |
| STATUS **KnlGetMemUsed**(TASK_ID taskId) |
| STATUS **KnlGetOSVersion**(P_U32 major, P_U32 minor) |
| STATUS **KnlGetPriority**(TASK_ID taskId) |
| STATUS **KnlGetStackInfo**(TASK_ID taskId, P_VOID *start, P_VOID *end) |
| U32 **KnlGetStatus**(TASK_ID taskId) |
| TASK_ID **KnlGetTaskId**(VOID) |
| U8 **KnlIsInIrpt**(VOID) |
| U8 **KnlIsSemaId**(SEMA_ID semaId) |
| U8 **KnlIsTaskId**(TASK_ID taskId); |
| STATUS **KnlResume**(TASK_ID taskId) |
| STATUS **KnlSetEventPort**(TASK_ID taskId, EVTPORT_ID portId) |
| STATUS **KnlSetSemaphore**(SEMA_ID semaId, U16 max, U16 init, BOOL fifo) |
| STATUS **KnlSignalSemaphore**(SEMA_ID semaId) |
| STATUS **KnlSuspend**(TASK_ID taskId) |
| STATUS **KnlSuspendFor**(TICK milliseconds, SWT_ID swtId) |
| STATUS **KnlWaitSemaphore**(SEMA_ID semaId, TICK milliseconds) |
| STATUS **KnlYield**(VOID) |
| |
| **IrDA Services** |
| STATUS **IrdSetDeviceInfo**(P_U8 info, U8 len) |
| STATUS **IrdInit**(U8 port) |
| STATUS **IrdDeInit**(void) |
| STATUS **IrdSetMaxTurnAroundTime**(U8 MaxTat) |
| STATUS **IrdInitTransceiver**(void) |
| STATUS **IrdShutDownTransceiver**(void) |
| STATUS **IrcClose**(void) |

| Function Declaration |
|---|
| void **IrcGetFormat**(P_U32 format) |
| void **IrcGetStatusEventCause**(P_U16 eventCause) |
| BOOL **IrcIsDeviceBusy**(void) |
| STATUS **IrcOpen**(TASK_ID AppCallback) |
| U16 **IrcRead**(P_U8 buff, U16 len) |
| void **IrcSetFormat**(U32 format) |
| STATUS **IrcWrite**(P_U8 buff, U16 len) |
| STATUS **ObxInit** (TASK_ID AppCallback) |
| STATUS **ObxDeinit**(void) |
| STATUS **ObxSaveInbox** (P_U8 buff, U16 len) |
| STATUS **ObxGetInboxLen** (void ) |
| STATUS **ObxSaveName** (P_U8 buff, U8 len) |
| U8 **ObxGetNameLen** (void) |
| STATUS **ObxPutOutbox** (P_U8 buff, U16 len) |
| void **ObxAbort**() |
| STATUS **ObxConnect**(void) |
| STATUS **ObxConReq**(void) |
| STATUS **ObxClientDisconnect**(void) |
| STATUS **ObxDiscReq**(BOOL Force) |
| STATUS **ObxPut**(void) |
| ObxAbortReason **ObxGetAbortReason**(void) |
| BOOL **ObxHeaderBuildUnicode** (ObxHeaderType Type, P_U8 Value, U16 Len) |
| BOOL **ObxHeaderBuild4Byte** (ObxHeaderType Type, U32 Value) |
| |
| **Memory Management Services** |
| STATUS **MemAddRegion**(P_MEM_REGION_ID pRegId, P_U32 startAddr, U32 endAddr) |
| P_VOID **MemCalloc**(U32 size); |
| P_VOID **MemCallocFrom**( MEM_REGION_ID regionId, U32 size, TASK_ID taskId); |
| STATUS **MemCopy**(P_U8 src_addr, P_U8 dest_addr, U32 size); |

| Function Declaration |
|---|
| STATUS **MemDelRegion**(MEM_REGION_ID regionId) |
| void **MemFree**( P_VOID pUsedMem ); |
| STATUS **MemGetAvailSize**(MEM_REGION_ID regionId, P_U32 pSizeAvail); |
| S32 **MemGetAvailStack**( void ); |
| STATUS **MemGetOrgRegionSize**(MEM_REGION_ID regionId, P_U32 pSize); |
| STATUS **MemGetTaskUsed**( TASK_ID taskId, P_U32 inSize); |
| U32 **MemGetTotalUsed**( void ); |
| P_VOID **MemMalloc**(U32 size); |
| P_VOID **MemMallocFrom**( MEM_REGION_ID regionId, U32 size, TASK_ID taskId); |
| P_VOID **MemRealloc**( P_VOID p, U32 size); |
| STATUS **MemResetRegion**(MEM_REGION_ID regionId) |
| STATUS **MemResizeRegion(**MEM_REGION_ID regionId, U32 endAddr) |
| |
| **Networking Services** |
| **Networking Configuration and Setup Services** |
| STATUS **NetInit**(void) |
| VOID **NetDeinit**(void) |
| VOID **NetConfigDNS**(P_S8 ipAddress, U8 which) |
| VOID **NetConfigGateway**(P_S8 ipAddress) |
| VOID **NetConfigISPAccount**(P_S8 UserID, P_S8 Password, P_U8 PhoneNum) |
| VOID **NetConfigLocalHostIP**(P_S8 ipAddress) |
| VOID **NetConfigMachineName**(P_S8 Name) |
| VOID **NetConfigModem**(P_S8 comPort,NET_DRIVER comDriver, P_S8 baudRate) |
| VOID **NetConfigPPP**(P_S8 UserID, P_S8 Password) |
| STATUS **NetDNSResolve**(P_S8 Fullname, IP_ADDR* iidp) |
| **Transport Services** |
| S32 **accept**(S32 S, `Struct sockaddr*` Name, P_S32 NameLen) |
| S32 **bind**(S32 S, `Struct sockaddr*` Name, S32 NameLen) |
| S32 **closesocket**(S32 S) |
| S32 **connect**(S32 S, `Struct sockaddr*` Name, S32 NameLen) |

| Function Declaration |
|---|
| S32 **socket**(S32 Domain, S32 Type, S32 Protocol) |
| S32 **fcntlsocket**(S32 S, S32 Cmd, S32 Arg) |
| Struct hostent* **gethostbyaddr_r**(P_S8 addr, S32 Len, S32 Type, struct hostent* result, P_S8 Buffer, S32 BufLen, P_S32 H_errnop) |
| Struct hostent* **gethostbyname_r**(P_S8 hnp, struct hostent* result, P_S8 buffer, S32 buflen, P_S32 h_errnop) |
| S32 **getpeername**(S32 S, `Struct sockaddr*` peer, P_S32 addrLen) |
| S32 **getsockname**(S32 S, `Struct sockaddr*` Name, P_S32 NameLen) |
| S32 **getsockopt**(S32 S, S32 level, S32 Optname, P_S8 Optval, P_S32 Optlen) |
| U32 **htonl**(U32 Val) |
| U16 **htons**(U16 Val) |
| U32 **iaddr**(Const P_S8 Dotted) |
| P_S8 **intoa**(Struct in_addr Addr) |
| S32 **ioctlsocket**(S32 s, S32 Request, P_S32 Arg) |
| S32 **listen**(S32 S, S32 Backlog) |
| U32 **ntohI**(U32 Val) |
| U16 **ntohs**(U16 Val) |
| S32 **readsocket**(S32 S, P_S8 buf, S32 Len) |
| S32 **recv**(S32 S, P_S8 buf, S32 Len, S32 Flags) |
| S32 **revfrom**(S32 S, P_S8 buf, S32 len, S32 flags, `Struct sockaddr*` from, P_S32 fromLen) |
| S32 **recvmsg**(S32 S, `Struct msghdr*` Msg, S32 Flags) |
| S32 **selectsocket**(S32 Nfds, Fd_set* Readfds, Fd_set* Writefds, Fd_set* Exceptfds, Struct timeval* Timeout) |
| S32 **send**(S32 S, P_S8 Buf, S32 Len, S32 Flags) |
| S32 **sendmsg**(S32 S, Struct msghdr* msg, S32 flags) |
| S32 **sendto**(S32 s, P_S8 buf, S32 len, S32 flags, struct sockaddr* to, S32 toLen) |
| S32 setsockopt(S32 S, S32 Level, S32 Optname, P_S8 Optval, S32 Optlen) |
| S32 **shutdown**(S32 S, S32 How) |
| S32 **writesocket**(S32 S, P_S8 Buf, S32 Len) |
| |

| Function Declaration |
|---|
| **Pen Input Handling Services** |
| STATUS **PenAddAreaToIC**(IC_ID icId, AREA_ID areaId); |
| STATUS **PenBindTaskToIC**(TASK_ID taskId, IC_ID icId); |
| STATUS **PenBringAreaBack**(AREA_ID areaId); |
| STATUS **PenBringAreaBackward**(AREA_ID areaId); |
| STATUS **PenBringAreaForward**(AREA_ID areaId); |
| STATUS **PenBringAreaFront**(AREA_ID areaId); |
| STATUS **PenCalibrate**(U8 logoFlag); |
| STATUS **PenCreateArea**(P_AREA_ID pAreaId); |
| STATUS **PenCreateIC**(P_IC_ID pIC); |
| STATUS **PenDeleteArea**(AREA_ID areaId); |
| STATUS **PenDeleteIC**(IC_ID icId, U8 deleteAllArea); |
| STATUS **PenGetAreaIdFromEvent**(P_EVENT event, P_AREA_ID pAreaId); |
| STATUS **PenGetAreaMode**(AREA_ID areaId, P_AREA_MODE pMode); |
| STATUS **PenGetAreaPos**(AREA_ID areaId, P_S16 pXSrc, P_S16 pYSrc, P_S16 pXDest, P_S16 pYDest); |
| STATUS **PenGetAreaType**(AREA_ID areaId, P_AREA_TYPE pType); |
| STATUS **PenGetEchoMode**(AREA_ID areaId, P_U8 pEchoMode); |
| STATUS **PenGetICFromArea**(AREA_ID areaId, P_IC_ID pIcId); |
| STATUS **PenGetInputTimeout**(IC_ID icId, P_TICK pInputTimeout); |
| STATUS **PenGetPenColor**(IC_ID icId, P_COLOR pPenColor); |
| STATUS **PenGetPenSize**(IC_ID icId, P_U8 pPenSize); |
| STATUS **PenGetPos**( P_S16 pX, P_S16 pY); |
| STATUS **PenGetPosFromEvent**( P_EVENT event, P_POINT *pPoints, P_U16 numberOfPoints); |
| VOID **PenGetSample**(); |
| STATUS **PenGetSamplingRate**(IC_ID icId, P_PEN_RATE pPenRate); |
| STATUS **PenInitArea**(AREA_ID areaId, S16 xSrc, S16 ySrc, S16 xDest, S16 yDest, U8 type, U8 mode, U8 panPosFlag, U8 echoMode); |
| STATUS **PenInitIC**(IC_ID icId, TASK_ID taskId, TICK penInputTimeout, PEN_RATE samplingRate, U8 iconScan, U8 penSize, COLOR penColor); |

| Function Declaration |
|---|
| U16 **PenMapX**(U16 x); |
| U16 **PenMapY**(U16 y); |
| STATUS **PenMoveAreaToTop**(IC_ID icId, AREA_ID areaId); |
| STATUS **PenRemoveAreaFromIC**(AREA_ID areaId); |
| STATUS **PenSetAreaMode**(AREA_ID areaId, AREA_MODE mode); |
| STATUS **PenSetAreaPos**(AREA_ID areaId, S16 xSrc, S16 ySrc, S16 xDest, S16 yDest); |
| STATUS **PenSetAreaType**(AREA_ID areaId, AREA_TYPE type); |
| STATUS **PenSetEchoMode**(AREA_ID areaId, U8 echoMode); |
| STATUS **PenSetInputTimeout**(IC_ID icId, TICK time); |
| STATUS **PenSetPenColor**(IC_ID icId, COLOR penColor); |
| STATUS **PenSetPenSize**(IC_ID icId, U8 penSize); |
| STATUS **PenSetRingBuffer**(APP_ID appId, U16 bufferSize); |
| STATUS **PenSetSamplingRate**(IC_ID icId, PEN_RATE rate); |

| Power Management Services | |
|---|---|
| STATUS | **PwrDisnotifyDoze**(TASK_ID taskId); |
| U32 | **PwrGetDeviceStatus**(VOID); |
| U32 | **PwrGetEnterSleep**(VOID); |
| U32 | **PwrGetEnterDoze**(VOID); |
| U32 | **PwrGetExitDoze**(VOID); |
| U32 | **PwrGetExitSleep**(VOID); |
| U32 | **PwrGetIdle**(VOID); |
| U16 | **PwrGetIdleTime**(VOID); |
| POWERMODE | **PwrGetMode**(VOID); |
| U32 | **PwrGetSysClk**(VOID); |
| U8 | **PwrIsIdleEnable**(VOID); |
| STATUS | **PwrNotifyDoze**(TASK_ID taskId); |
| STATUS | **PwrRestartIdle**(VOID); |
| VOID | **PwrSetEnterDoze**(U32 device); |
| VOID | **PwrSetEnterSleep**(U32 device); |

| Function Declaration |
|---|
| VOID **PwrSetExitDoze**(U32 device); |
| VOID **PwrSetExitSleep**(U32 device); |
| VOID **PwrSetIdle**(U32 device); |
| STATUS **PwrSetIdleTime**(U16 second); |
| STATUS **PwrSetMode**(POWERMODE mode) |
| STATUS **PwrStopIdle**(VOID); |
| |
| **RTC Handling Services** |
| STATUS **RtcSetTime**(U8 hour, U8 minute, U8 second); |
| STATUS **RtcGetTime**(P_U8 hour, P_U8 minute, P_U8 second); |
| STATUS **RtcSetDate**(U16 year, U8 month, U8 day); |
| STATUS **RtcGetDate**(P_U16 year, P_U8 month, P_U8 day); |
| STATUS **RtcGetDayofWeek**(U16 year, U8 month, U8 day, P_U8 dayofweek); |
| void **RtcIsLeapYear**(U16 year, P_U8 leapyear); |
| STATUS **RtcValidDate**(U16 year, U8 month, U8 day); |
| STATUS **RtcValidTime**(U8 hour, U8 minute, U8 second); |
| STATUS **RtcSetDateTime**(U16 year, U8 month, U8 day, U8 hour, U8 minute, U8 second); |
| STATUS **RtcGetDateTime**(P_U16 year, P_U8 month, P_U8 day, P_U8 hour, P_U8 minute, P_U8 second); |
| STATUS **RtcGetGMTime**(P_U16 year, P_U8 month, P_U8 day, P_U8 hour, P_U8 minute, P_U8 second) |
| STATUS **RtcGetGMTOffset**(P_S8 Gmtoffset) |
| STATUS **RtcSetGMTOffset**(S8 Gmtoffset) |
| |
| **Sci Management Services** |
| STATUS **SciBindPort**(SCI_PORT_ID portId, U8 uartPort); |
| STATUS **SciClose**( SCI_PORT_ID portId); |
| STATUS **SciConfig**(SCI_PORT_ID portId, U8 mode, U8 baudRate, U8 parity, U8 stopBits, U8 charLen); |
| STATUS **SciCreate**(P_SCI_PORT_ID portId); |

| Function Declaration |
|---|
| STATUS **SciCtsStatus**(SCI_PORT_ID portId); |
| STATUS **SciFlowCtrl**(SCI_PORT_ID portId, U8 controlType); |
| STATUS **SciFlushFifo**(SCI_PORT_ID portId, U8 fifoFlag); |
| STATUS **SciGetConfig**(SCI_PORT_ID portId, P_U8 pMode, P_U32 pBaudRate, P_U8 pParity, P_U8 pStopBits, P_U8 pCharLen); |
| VOID **SciGetData**(P_U16 pData); |
| STATUS **SciOpen**(SCI_PORT_ID portId); |
| STATUS **SciRcvCtrl**(SCI_PORT_ID portId, U8 controlType); |
| STATUS **SciReadData**(SCI_PORT_ID portId, P_U8 pData, U16 bufSize, P_U16 pSizeRead); |
| STATUS **SciReceive**(SCI_PORT_ID portId, U8 receiveFlag); |
| STATUS **SciRtsStatus**(SCI_PORT_ID portId); |
| STATUS **SciSend**(SCI_PORT_ID portId, U8 sendFlag, P_U8 pData, U32 dataLen); |
| STATUS **SciSendAbort**(SCI_PORT_ID portId, U8 abortFlag, P_U8 *ppSendData, P_U32 sendSize); |
| STATUS **SciSendCtrl**(SCI_PORT_ID portId, U8 controlType); |
| STATUS **SciSetDelay**(SCI_PORT_ID portId, U8 type, U16 delay); |
| STATUS **SciSetFifoLevel**(SCI_PORT_ID portId, U8 fifoFlag, U8 fifoLevel); |
| STATUS **SciSetRxBufSize**(SCI_PORT_ID portId, U16 newSize); |
| STATUS **SciSetTargetTask**(SCI_PORT_ID portId, TASK_ID taskId); |
| void **SciSetTimeout**(P_SCI_TMOUT pTmout, TICK timeout, EVTTYPE portId) |
| STATUS **SciUnbindPort**(SCI_PORT_ID portId); |
| |
| **Software Keyboard Services** |
| STATUS **SkyBind**( SKY_ID skyId, APP_ID appId); |
| STATUS **SkyClose**( SKY_ID skyId); |
| STATUS **SkyCreate**(SKY_ID skyId); |
| STATUS **SkyOpen**( SKY_ID skyId); |
| STATUS **SkyOpenDefKB**( SKY_ID skyId, U16 xPos, U16 yPos); |
| STATUS **SkyOpenKB**( SKY_ID skyId, U16 xPos, U16 yPos, U16 keyWidth, U16 keyHeight, U16 numCol, U16 numRow, P_U16 pKeyMap, P_U8 bitmap); |

| Function Declaration |
|---|
| VOID **SkyReadKey**(P_U16 pKey); |
| STATUS **SkySetAutoRepeat**(SKY_ID skyId, U16 beginTime, U16 repeatTime); |
| STATUS **SkySetKeyMap**( SKY_ID skyId, P_U16 keyMap, P_U8 keyBmp, U16 bmpWt, U16 bmpHt); |
| STATUS **SkySetKeySize**( SKY_ID skyId, U16 keyWt, U16 keyHt); |
| STATUS **SkySetOrigin**( SKY_ID skyId, U16 xPos, U16 yPos); |
| |
| **Software Timer Handling Services** |
| SWT_ID **SwtCreate**(VOID); |
| STATUS **SwtDelete**(SWT_ID swtId); |
| TICK **SwtDiffRefTime**(TICK beginTime, TICK endTime) |
| STATUS **SwtGetCount**(SWT_ID swtId, P_TICK pCount) |
| P_EVENT **SwtGetEvent**(SWT_ID swtId, P_U16 pSize) |
| U32 **SwtGetResolution**(VOID); |
| STATUS **SwtGetTaskId**(SWT_ID swtId, P_TASK_ID pTaskId); |
| STATUS **SwtInitTimer**(SWT_ID swtId, TASK_ID taskId, TICK count, TICK reload, P_EVENT pEvent, U16 size, P_VOID func, U32 arg) |
| U8 **SwtIsInUse**(SWT_ID swtId); |
| U8 **SwtIsSwtId**(SWT_ID swtId); |
| TICK **SwtReadRefTime**(VOID); |
| STATUS **SwtRestartTimer**(SWT_ID swtId, TICK count); |
| STATUS **SwtSetArg**(SWT_ID swtId, U32 arg); |
| STATUS **SwtSetCount**(SWT_ID swtId, TICK count); |
| STATUS **SwtSetEvent**(SWT_ID swtId, P_EVENT pEvent, U16 eventSize) |
| STATUS **SwtSetFunc**(SWT_ID swtId, P_VOID func, U32 arg) |
| STATUS **SwtSetTaskId**(SWT_ID swtId, TASK_ID taskId); |
| STATUS **SwtStartTimer**(SWT_ID swtId); |
| STATUS **SwtStopTimer**(SWT_ID swtId); |
| |
| **Text Management Services** |

| Function Declaration |
|---|
| STATUS **TxtCreateTmplt**(P_TMPLT_ID templateId); |
| STATUS **TxtDeleteTmplt**(TMPLT_ID templateId); |
| STATUS **TxtMap**(TMPLT_ID templateId, U8 bitLen, P_TEXT buffer, U16 size, P_U8 pNextLine); |
| STATUS **TxtPrintf**(TMPLT_ID templateId, P_U8 pFormatStr, P_VOID argList) |
| STATUS **TxtReadCurPos**(TMPLT_ID templateId, P_U16 cursor); |
| STATUS **TxtSetCurPos**(TMPLT_ID templateId, U16 cursor); |
| STATUS **TxtSetCurXY**(TMPLT_ID templateId, U16 xPos, U16 yPos); |
| STATUS **TxtSetFontColor**( TMPLT_ID templateId, COLOR fontColor); |
| STATUS **TxtSetFontStyle**( TMPLT_ID templateId, STYLE fontStyle); |
| STATUS **TxtSetFontType**( TMPLT_ID templateId, FONT_TYPE fontType); |
| STATUS **TxtSetLineWt**(TMPLT_ID templateId, U16 lineWt); |
| STATUS **TxtSetTmpltOrigin**(TMPLT_ID templateId, U16 xSrc, U16 ySrc); |
| STATUS **TxtSetTmpltSize**(TMPLT_ID templateId, U16 width, U16 height); |
| STATUS **TxtSetupTmplt**(TMPLT_ID templateId, FONT_TYPE fontType, STYLE outputStyle, COLOR greyLevel, U16 xPos, U16 yPos, U16 width, U16 height); |
| STATUS **TxtUnmap**(TMPLT_ID templateId); |

# Motorola

# PPSM-GT User Guide

# Version 1.1

## Credits

| | |
|---:|---|
| **writing lead:** | Jeffrey Chia |
| **other writers:** | Smita Manathkar |
| **engineering, documentation review:** | Casper Mok, Kent Ip, Alex Yu, Frank Ma, Sarah Chiu,  Eric Chan, Minna Lai, Jason Ma, Christina Ying, Leila He, Patrick Lam |
| **Document review & editing:** | Nicholas Evans |
| **Special Thanks** | John Roseborough, Mary Thomas |