PM5357

S/UNI-622-POS

SOFTWARE DRIVER FOR THE S/UNI-622-POS

PRELIMINARY

ISSUE 1

PROPRIETARY AND CONFIDENTIAL TO PMC-SIERRA, INC., AND FOR ITS CUSTOMERS' INTERNAL USE

PMC-Sierra, Inc. 105 - 8555 Baxter Place Burnaby, BC Canada V5A 4V7 604 .415.6000

PROPRIETARY AND CONFIDENTIAL TO PMC-SIERRA, INC., AND FOR ITS CUSTOMERS' INTERNAL USE

CONTENTS

PROPRIETARY AND CONFIDENTIAL TO PMC-SIERRA, INC., AND FOR ITS CUSTOMERS' INTERNAL USE

PROPRIETARY AND CONFIDENTIAL TO PMC-SIERRA, INC., AND FOR ITS CUSTOMERS' INTERNAL USE

1 OVERVIEW

1.1 Scope

A software driver for the S/UNI-622-POS (PM5357) is described in this document. The driver is written in C language and is structured such that it is reusable and can be ported to a user's environment with minimal modification. The application programmer's interface (API) and an example of how to operate the S/UNI-622-POS via this API is presented.

The driver is built and tested on the S/UNI-622-POS reference design[1]. The software driver interfaces to the MC68332 processor of the reference design board via processor dependent software, a simple software dispatcher and a serial port interface. A thorough description of this additional software is not covered in this document, nor is it supported, but the source code is made available to users with the driver source code.

This driver source code is preliminary and not fully tested at the time this document was issued. Please contact PMC for the latest status of the source code.

1.2 Audience

The intended audience for this document is Software Engineers that use this software to gain familiarity with the operation of the S/UNI-622-POS product, or to use this software in their own systems.

1.3 Objectives

The objective of making this driver available is to provide an example that may shorten the learning curve and development time that users require to write S/UNI-622-POS drivers for their own systems. It also allows use of the reference design as an evaluation/development vehicle, for users to gain familiarity with the S/UNI-622-POS and for users to code/unit test their own software while waiting for their own hardware prototypes – effectively allowing the user's hardware and software development to run in parallel.

PROPRIETARY AND CONFIDENTIAL TO PMC-SIERRA, INC., AND FOR ITS CUSTOMERS' INTERNAL USE

2 SOFTWARE DRIVER FEATURES

The following features are provided by the software driver:

- Driver supports multiple S/UNI-622-POS chips.
- Driver abstracts each S/UNI-622-POS into a logical device to provide access to these by device ID.
- Driver provides a device data block that allows the user to specify the configuration of logical devices.
- API routines are provided to reset, initialize or activate a logical device.
- Driver provides a header file that defines all registers and bit fields of the S/UNI-622-POS to reduce the coding effort. The defines are parsed from the datasheet and provide a means to access features of the S/UNI-622-POS directly by using the read/write register access routines of the API.
- API routines to poll and accumulate counter statistics
- An interrupt service routine to dispatch interrupt events to a user application
- Source code is written in ANSI C language.
- All source code that may need to be modified to port the driver to another environment is located within a separate file.
- All routines are re-entrant. The user can use semaphores to lock access to the device data block or the register space. Since the use of semaphores is dependent on the software environment it is left to the user to add them if necessary.
- Source code has debug macros that can be used during debug to dump error events to a console.

The following features are provided as a user interface to the reference design hardware:

- A serial port interface that accepts user commands and provides a log of events due to interrupts or accumulation of statistics.

PROPRIETARY AND CONFIDENTIAL TO PMC-SIERRA, INC., AND FOR ITS CUSTOMERS' INTERNAL USE

- Commands that invoke the API routines, and allow the API data structures to be accessed.

PROPRIETARY AND CONFIDENTIAL TO PMC-SIERRA, INC., AND FOR ITS CUSTOMERS' INTERNAL USE

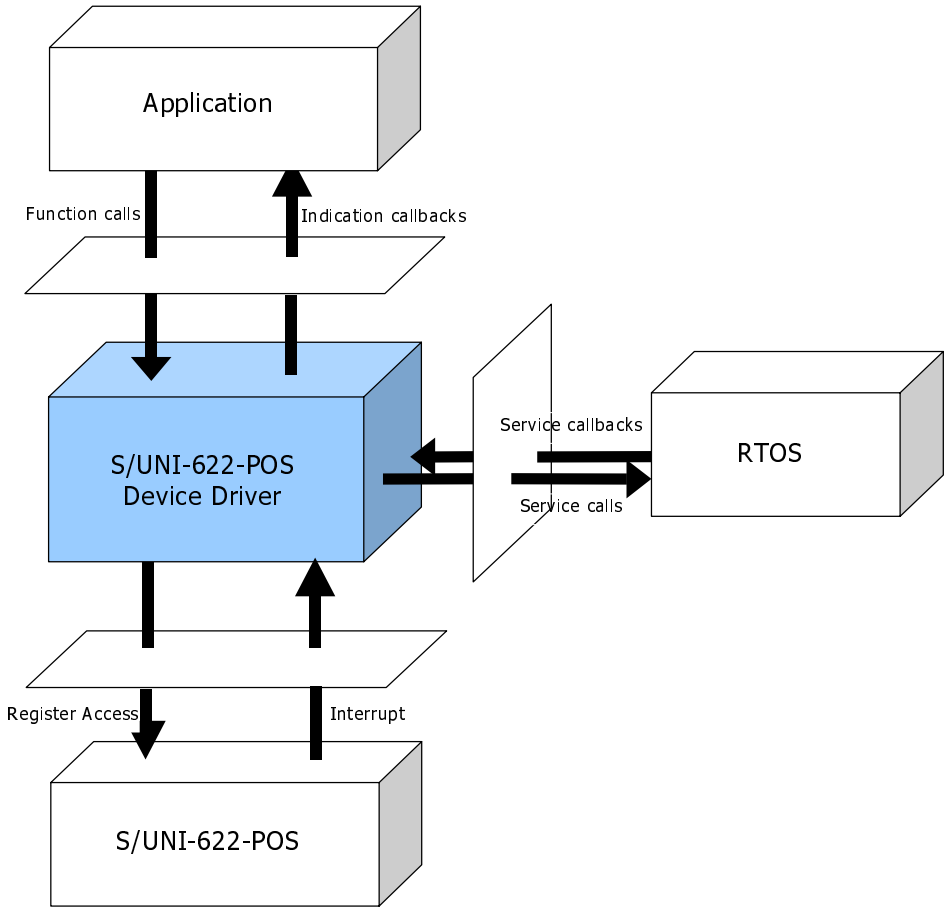
3 APPLICATION PROGRAMMER'S INTERFACE

3.1 Software Architecture

The S/UNI-622-POS driver interfaces to software and hardware components as shown in figure 1. Each of these interfaces is described in the following sections. Communication between components is via function calls, and in the opposite direction, via function callbacks.

The application programmer's interface (API) covers the function calls and data structures passed through the three planes shown in figure 1. These planes interface to the application component, the RTOS component, and the S/UNI-622-POS hardware. The arrows show the direction of a function call.

Figure 1. Interfaces of the S/UNI-622-POS Driver



PROPRIETARY AND CONFIDENTIAL TO PMC-SIERRA, INC., AND FOR ITS CUSTOMERS' INTERNAL USE

3.1.1 Application Interface

The user's application task interfaces to the driver by making function calls that command the driver to carry out a specified operation on the S/UNI-622-POS. The function prototypes are shown in section 3.5.

Each S/UNI-622-POS in a system is managed as a separate device with a unique data structure to manage it. For this reason the application must identify the device when making a driver function call with a device ID.

When an event occurs within the S/UNI-622-POS the driver must notify the user application that the event occurred. This is done via an Indication function callback with the device ID as a function parameter.

3.1.2 RTOS Interface

The real-time operating system (RTOS) provides the environment for the driver and user application to run. Typically this would be a multi-tasking, single processor, environment with semaphores to protect critical sections of code or variables from being corrupted. The RTOS provides the following services to the driver:

- Install a system timeout, or periodic timer.
- Install an interrupt service routine that is dispatched when a S/UNI-622-POS interrupt pin is active.
- Provide memory management services to map the register space, allocate data structures and translate between virtual and physical addressing.
- Creation and management of tasks.
- Task to task communication or message queues.
- Semaphores

The simplest environment for this driver to operate in is a single task where the application is tightly coupled to the driver. In this case one task (the application) directly calls the API routines and another task (the RTOS) may call the service callback. The service callback could be the interrupt service routine or a routine that periodically polls and accumulates counter statistics. In this case the driver is

PROPRIETARY AND CONFIDENTIAL TO PMC-SIERRA, INC., AND FOR ITS CUSTOMERS' INTERNAL USE

written such that there is no contention among the application task and the RTOS task for access to a S/UNI-622-POS register or a field within the device data block.

In a multi-tasking environment, where multiple applications may want to access the same device simultaneously, the user could provide a loosely coupled interface between the API and the application tasks. This would be implemented by a queueing mechanism and/or semaphores to block another application task from calling an API routine of the device until the current API call has completed.

3.1.3 S/UNI-622-POS Hardware Interface

The S/UNI-622-POS hardware interfaces to the driver via register access and via an interrupt pin.

3.2 Driver Files

The driver is designed for use with the reference design board[1] and some of the interfaces may need to be modified or ported to the user's environment. An implementation file and a header file with the functions and defines that may need to be modified to port the driver to another environment have been supplied in the files "pos_p.h" and "pos_p.c".

The registers and bit fields of the S/UNI-622-POS have taken from the datasheet[2] and placed in the header file "pos_r.h". Further datasheet related defines are provided in the file "pos_d.h".

The files "pos.h" and "pos.c" provide the rest of the driver.

An example application is provided by the file "app.c".

3.3 Using the API to Access Features of the S/UNI-622-POS

This software driver provides a framework for users to integrate the S/UNI-622-POS into their own systems. It provides an API of routines that all users of the S/UNI-622-POS would require. Additional features of the S/UNI-622-POS, such as access to overhead, diagnostics or configuration are available directly to the user via read/write functions of the API, and by using the header files that define all registers and bit fields.

PROPRIETARY AND CONFIDENTIAL TO PMC-SIERRA, INC., AND FOR ITS CUSTOMERS' INTERNAL USE

3.3.1 Access to Features via the Registers

For example, the following code segment shows how a user could modify the value of the outgoing Path Signal Label (C2 byte) for an unscrambled POS application and read the receive path signal label.

```
U8 Value;

DeviceId = DEVICE_ID_CHIP1;

/* modify the transmit path signal label for POS mode*/
posWriteRegister(DeviceId,REG_48_TPOP_Path_Signal_Label,0xCF);

/* read the receive path signal label */
posReadRegister(DeviceId,REG_37_RPOP_Path_Signal_Label,&Value);
```

3.3.2 Power-on Initialization, Self Test and Activation

A user will typically modify the “posEntryPoint(…)” function for their application environment and perform power-on initialization and self test (POST) of their system. For this reason the user may need to reset, initialize and activate the S/UNI-622-POS directly via the API to place it in an operational state following the POST. The following example illustrates this:

```
/* perform power-on initialization of the S/UNI-622-POS */
posEntryPoint();

/* power on self test : configure a diagnostic loopback */
DeviceId = DEVICE_SUNI_622_POS_1;
pDevice = posGetDevice(DeviceId);
pDevice->InitVector = NULL;
pDevice->ActivateVector = NULL;
posReset(DeviceId);
value = posReadRegister(DeviceId, REG_02_Master_Configuration_2);
value |= BIT_02_SDLE;
posWriteRegister(DeviceId,REG_02_Master_Configuration_2,value);
posInit(DeviceId);
posActivate(DeviceId);

/* finished POST so configure for normal operation with defaults */
posReset(DeviceId);
posInit(DeviceId);
posActivate(DeviceId);
```


PROPRIETARY AND CONFIDENTIAL TO PMC-SIERRA, INC., AND FOR ITS CUSTOMERS’ INTERNAL USE

3.3.3 Event Notification

Finally a user may need to be notified of an event within the S/UNI-622-POS. This occurs through the driver function “posDispatchEvent(..)”. For example, the user may wish to be notified that the receive path signal label has changed. The user must first enable the interrupt event as shown here before device activation, or within the “ActivateVector” before the device was activated:

```
/* get the current enables, without clearing status */
/* assume (EXTD=0) in register 0x36 */
posReadRegister(DeviceId,REG_33_RPOP_Interrupt_Enable,&Value);

/* modify the interrupt enable bit */
posWriteRegister(DeviceId,REG_33_RPOP_Interrupt_Enable,(Value|BIT_33_PSLE));
```

At some time later when the C2 byte changes the driver’s interrupt service routine would dispatch the PSLI interrupt to the “posDispatchEvent(…)” function which is shown below:

```
void posDispatchEvent(int DeviceId, int regnum, U8 val)
{
    char msg[MAX_MESSAGE_LENGTH];
    DebugMsg(msg,“Device%i: Int reg 0x%02lX = 0x%02lX”,DeviceId,regnum,val);
}
```

In this driver the event is simply written to the console, but the user would need to port this function to interface to the user’s application, via a task communication mechanism provided by the RTOS. In this driver all interrupt events are assumed to have equal priority and are written to the console as they are observed in the interrupt service routine. A user’s application would want to prioritize and process these interrupt events. Typically the processed interrupt events will be dispatched for storage in a database of the application.

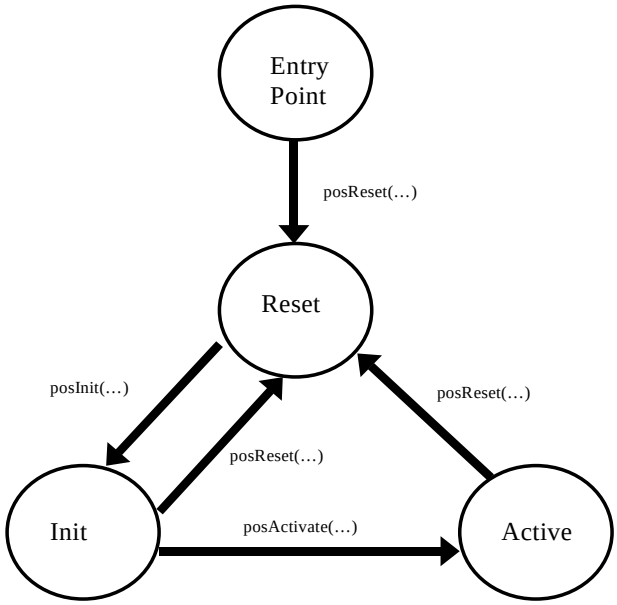
3.4 Data Structures

3.4.1 tPosState

The driver maintains one device per S/UNI-622-POS. Operation of a device is maintained via a device data block which is defined in the following sections. The device data block has a state variable that is used to manage the state transitions of the device as shown in figure 2.

PROPRIETARY AND CONFIDENTIAL TO PMC-SIERRA, INC., AND FOR ITS CUSTOMERS’ INTERNAL USE

Figure 2. State Diagram of a Device



The states of a device are defined as follows:

posENTRY_POINT: This is the default state when the variable has not yet been assigned by an API call that resets, initializes or activates the device.

posRESET: The device has been reset via software. No other register accesses have been performed after the reset. This state must occur before the device can be initialized and ensures the device is in a known state of operation. Specifically the device is idle while in the RESET state and does not affect operation of the system.

posINIT: The device data block (DDB) holds the initialization of the device and and is passed into the posInit(…) function to place the device in the initialization state. The registers of the device are initialized but the device does not yet interact with other system components. (ie. the device will not activate the interrupt pin, or otherwise interact with the system.)

posACTIVE: The device data block holds the interrupt enables and other information necessary to allow the device to interact with the system. This information is passed in the function call posActivate(…) to place the device in the active state. In the active state the S/UNI-622-POS would be able to activate

PROPRIETARY AND CONFIDENTIAL TO PMC-SIERRA, INC., AND FOR ITS CUSTOMERS' INTERNAL USE

the interrupt pin and interact with the other hardware components of the system. The user must ensure that the processor has assigned an interrupt service routine and has activated other system components as necessary before placing a device in the active state.

3.4.2 posDDB

The device context for the S/UNI-622-POS is provided in the device data block which has the members shown below. This structure is allocated within the function call posEntryPoint(…).

Member	Description
`u32 BaseAddress`	Address of the S/UNI-622-POS in memory space
`tPosState State`	State of this chip device
`tPosRegisterValue* InitVector`	Specifies initialization of chip device. Assigned a NULL value for the chip defaults.
`tPosRegisterValue* ActivateVector`	Specifies interrupt enables for chip device. Assigned a NULL value for chip defaults (all interrupts disabled).
`u8 XXXXIntEn_RR`	There are 18 interrupt enable registers associated with a channel device. “XXXX” represents the hardware block and “RR” represents the register number.
`u8 XXXXIntMask_RR`	There are 18 interrupt maskable registers associated with a channel device. These specify the interrupt status bits to check within the interrupt service routine. “XXXX” represents the hardware block and “RR” represents the register number.
`u32 XXXXCount`	There are 20 counters associated with a device. Here “XXXX” represents the

PROPRIETARY AND CONFIDENTIAL TO PMC-SIERRA, INC., AND FOR ITS CUSTOMERS’ INTERNAL USE

error counter name.

3.5 Application Interface Function Prototypes

The API has the following functions that allow the application component to request an action from the driver:

3.5.1 posEntryPoint

Description:	This is the first API call that should be made. It allocates and assigns devices for all S/UNI-622-POS chips in the system. Then it resets, initializes and finally activates the devices.
Function Prototype:	**`void posEntryPoint(void)`**
Function Parameters:	**none**
Return Value:	**`posSUCCESS`** – all devices have been successfully placed in the active state **`posFAILURE`** – one or more devices could not be activated.
Additional Notes:	This function is in the porting file “pos_p.c” because it needs to be modified for the users environment.

3.5.2 posExitPoint

Description:	This function does the reverse of the PosEntryPoint function. It places all devices in the reset state and then deallocates all device resources.
Function Prototype:	**`void posExitPoint(void)`**
Function Parameters:	**none**
Return Value:	**none**
Additional Notes:	This function is in the porting file “pos_p.c” because it needs to be modified for the users environment.

PROPRIETARY AND CONFIDENTIAL TO PMC-SIERRA, INC., AND FOR ITS CUSTOMERS’ INTERNAL USE

3.5.3 posReset

Description: This function performs a software reset of a S/UNI-622-POS device.

Function Prototype: **`tPosStatus posReset(int DeviceId)`**

Function Parameters: **`DeviceId`** – specifies the device data block

Return Value: **`posSUCCESS`** – reset completed

`posFAILURE` – invalid device identifier specified.

Additional Notes: none

3.5.4 posInit

Description: This function initializes the device, but does not enable interrupts.

Function Prototype: **`tPosStatus posInit(int DeviceId)`**

Function Parameters: **`DeviceId`** – specifies the device data block for the device.

Return Value: **`posSUCCESS`** – the device has been successfully initialized.

`posFAILURE` – invalid device ID

Additional Notes: The caller must ensure the device data block specifies the initialization values for the device.

3.5.5 posActivate

Description: This function places the device in an active state by enabling device interrupts.

PROPRIETARY AND CONFIDENTIAL TO PMC-SIERRA, INC., AND FOR ITS CUSTOMERS' INTERNAL USE

Function Prototype: **`tPosStatus posActivate(int DeviceId)`**

Function Parameters: **`DeviceId`** – specifies the device data block for the device.

Return Value: **`posSUCCESS`** – the device has been successfully activated.

`posFAILURE` – invalid device ID

Additional Notes:

3.5.6 posIsr

Description: This function is the interrupt service routine for the chip device.

Function Prototype: **`void posIsr(int DeviceId)`**

Function Parameters: **`DeviceId`** – specifies the device data block for the chip device.

Return Value: **`posSUCCESS`** – the interrupt has been processed.

`posFAILURE` – invalid device ID or no Interrupts were active.

Additional Notes: Only the chip device requires an ISR. The channel devices are serviced by the chip's ISR.

3.5.7 posEnableInterrupts

Description: This function re-enables interrupts for the device while the device is in the ACTIVE state.

Function Prototype: **`void posEnableInterrupts(int DeviceId)`**

Function Parameters: **`DeviceId`** – specifies the device data block.

PROPRIETARY AND CONFIDENTIAL TO PMC-SIERRA, INC., AND FOR ITS CUSTOMERS' INTERNAL USE

Return Value: **none**

Additional Notes: This call is only valid in the active state.

3.5.8 posDisableInterrupts

Description: This function disables interrupts for the device while the device is in the ACTIVE state.

Function Prototype: **`void posDisableInterrupts(int DeviceId)`**

Function Parameters: **`DeviceId`** – specifies the device data block for the device.

Return Value: **none**

Additional Notes: This call is only valid in the active state.

3.5.9 posStatistics

Description: This function should be called periodically to accumulate counter statistics.

Function Prototype: **`void posStatistics(int DeviceId)`**

Function Parameters: **`DeviceId`** – specifies the device data block for the device.

Return Value: none

Additional Notes:

3.5.10 posClearCounters

Description: This function clears the accumulated counter values.

Function Prototype: **`void posClearCounters(int DeviceId)`**

Function Parameters: **`DeviceId`** – specifies the device data block for the

PROPRIETARY AND CONFIDENTIAL TO PMC-SIERRA, INC., AND FOR ITS CUSTOMERS' INTERNAL USE

device.

Return Value: none

Additional Notes:

3.5.11 posReadRegister

Description: This function reads the S/UNI-622-POS register associated with the device.

Function Prototype:
```
bool posReadRegister(int DeviceId,
int offset,U8* value)
```

Function Parameters: **DeviceId** – specifies the device data block for the device.

offset – specifies the register number from the datasheet.

value – is a pointer to an 8-bit value which is modified with the value read.

Return Value: **true** – the register was successfully read

false – the register could not be read

Additional Notes: The device should be in the reset, init or active state when making this call.

3.5.12 posWriteRegister

Description: This function writes the S/UNI-622-POS register associated with the device.

Function Prototype:
```
bool posWriteRegister(int DeviceId,
int offset,U8 value)
```

Function Parameters: **DeviceId** – specifies the device data block for the device.

offset – specifies the register number from the datasheet.

PROPRIETARY AND CONFIDENTIAL TO PMC-SIERRA, INC., AND FOR ITS CUSTOMERS' INTERNAL USE

value – is the 8-bit value to write.

Return Value: **true** – the register was successfully written

false – the register could not be written

Additional Notes: The device should be in the reset, init or active state when making this call.

3.5.13 posReadSstb

Description: This function reads the S/UNI-622-POS SSTB internal register associated with the device.

Function Prototype:
```
bool posReadSstb(int DeviceId,
int offset,U8* value)
```

Function Parameters: **DeviceId** – specifies the device data block for the device.

offset – specifies the indirect address as shown in the datasheet.

value – is a pointer to an 8-bit value which is modified with the value read.

Return Value: **true** – the register was successfully read

false – the register could not be read

Additional Notes: The device should be in the reset, init or active state when making this call.

3.5.14 posWriteSstb

Description: This function writes the S/UNI-622-POS SSTB internal register associated with the device.

Function Prototype:
```
bool posWriteRegister(int DeviceId,
int offset,U8 value)
```

Function Parameters: **DeviceId** – specifies the device data block for the device.

PROPRIETARY AND CONFIDENTIAL TO PMC-SIERRA, INC., AND FOR ITS CUSTOMERS' INTERNAL USE

offset – specifies the indirect address as shown in the datasheet.

value – is the 8-bit value to write.

Return Value: **true** – the register was successfully written

false – the register could not be written

Additional Notes: The device should be in the reset, init or active state when making this call.

3.5.15 posReadSptb

Description: This function reads the S/UNI-622-POS SPTB internal register associated with the device.

Function Prototype:
```
bool posReadRegister(int DeviceId,
int offset,U8* value)
```

Function Parameters: **DeviceId** – specifies the device data block for the device.

offset – specifies the indirect address as shown in the datasheet.

value – is a pointer to an 8-bit value which is modified with the value read.

Return Value: **true** – the register was successfully read

false – the register could not be read

Additional Notes: The device should be in the reset, init or active state when making this call.

3.5.16 posWriteSptb

Description: This function writes the S/UNI-622-POS SPTB internal register associated with the device.

Function Prototype:
```
bool posWriteRegister(int DeviceId,
int offset,U8 value)
```


PROPRIETARY AND CONFIDENTIAL TO PMC-SIERRA, INC., AND FOR ITS CUSTOMERS' INTERNAL USE

Function Parameters: **`DeviceId`** – specifies the device data block for the device.

`offset` – specifies the indirect address as shown in the datasheet.

`value` – is the 8-bit value to write.

Return Value: **`true`** – the register was successfully written

`false` – the register could not be written

Additional Notes: The device should be in the reset, init or active state when making this call.

3.6 RTOS Interface Function Prototypes

The driver requests services from the RTOS which are defined in the following function prototypes:

3.6.1 InstallIsr

Description: This function requests the RTOS to install an interrupt service routine.

Function Prototype: **`void InstallIsr(void func(void*),void* context)`**

Function Parameters: **`func`** – is the function which is to be called when an interrupt occurs. (ie. **`posIsr`**)

`context` – specifies the context passed as a parameter in the function call. (ie. the device ID)

Return Value: none

Additional Notes:

3.6.2 InstallTimer

Description: This function requests the RTOS to periodically call a

PROPRIETARY AND CONFIDENTIAL TO PMC-SIERRA, INC., AND FOR ITS CUSTOMERS' INTERNAL USE

timer function.

Function Prototype: **`void InstallTimer(int msec, (void) *func(int), void* context)`**

Function Parameters: **`msec`** – is the period of the timer in milliseconds.

`func` – is the function which is to be called periodically. (ie. **`posStatistics`**)

`context` – specifies the context passed as a parameter in the function call. (ie. the device ID)

Return Value: none

Additional Notes:

3.7 S/UNI-622-POS Interface Function Prototypes

The driver uses macros to read and to write the hardware registers. These need to be modified to port the driver to a different environment. The macro definitions are as follows, where “c” is the device data block pointer, “regnum” is the register, and “value” is the 8-bit value written to a register:

```
#define posWrite(c, regnum, value) (*(U8*)( c->BaseAddress + (regnum) ) = (value))

#define posRead(c, regnum) (*(U8*)( c->BaseAddress + (regnum) ))
```


PROPRIETARY AND CONFIDENTIAL TO PMC-SIERRA, INC., AND FOR ITS CUSTOMERS’ INTERNAL USE

4 APPENDIX A. SOURCE CODE

Please contact PMC-Sierra to obtain the source code.

PROPRIETARY AND CONFIDENTIAL TO PMC-SIERRA, INC., AND FOR ITS CUSTOMERS' INTERNAL USE

5 REFERENCES

[1] PMC-981070, S/UNI-622-POS Reference Design, PMC-Sierra, Issue 1.
[2] PMC-980911, S/UNI-622-POS Datasheet, PMC-Sierra, Issue 1, August 1998.

PROPRIETARY AND CONFIDENTIAL TO PMC-SIERRA, INC., AND FOR ITS CUSTOMERS' INTERNAL USE

6 SOFTWARE CUSTOMER FEEDBACK FORM

At PMC-Sierra we are always evaluating the ways we support our customers to improve their time-to-market and success in using our products.
We encourage you to take the time to fill out the following feedback form and fax it back to the applications group at **604-415-6206**. Alternatively you can e-mail the information to apps@pmc-sierra.com . Thank you in advance for you're valuable feedback.

Please provide you're name, company and contact information below:

Name:_____

Company:_____

Contact phone and/or e-mail: _____

SECTION 1. QUESTIONS
Please mark your responses to the following questions in the table below:

1. Have you previously used software source code provided for free by other integrated circuit vendors?	Yes	No
2. Has the availability of this software affected your decision to choose PMC-Sierra as a vendor of integrated circuits?	Yes	No
3. Where did you first hear about PMC-Sierra's software driver source code?		
a) PMC-Sierra Web site (www.pmc-sierra.com)		
b) Co-workers		
c) PMC-Sierra Salesperson or Sales Representative		
d) PMC-Sierra Customer Support Engineer		
e) Interop trade show		
f) SATURN conference		
g) Other (please specify _____)		
4. Did this software driver reduce your time-to-market or shorten the learning curve of using a PMC-Sierra integrated circuit?	Yes	No
a) Estimated time saved: _____		
5. Do you feel that another vendor of integrated circuits provides better software support for their integrated ciruit?	Yes	No
If you answered yes then please specify the vendor, the product, and the software support the vendor provided		

PROPRIETARY AND CONFIDENTIAL TO PMC-SIERRA, INC., AND FOR ITS CUSTOMERS' INTERNAL USE

below.

_____

_____

SECTION 2. RATING

Please rate this software source code in the following areas:
(1 = outstanding, 2 = exceeded expectations, 3 = met expectations, 4 = inadequate, 5 = poor)

	RATING				
Document:					
Readability	1	2	3	4	5
Targeted to audience	1	2	3	4	5
Usefulness	1	2	3	4	5
Completeness	1	2	3	4	5
Detailed	1	2	3	4	5
Accuracy	1	2	3	4	5
Application Programmers' Interface (API):					
Completeness	1	2	3	4	5
Usability	1	2	3	4	5
Source Code:					
Structure	1	2	3	4	5
Style	1	2	3	4	5
Usefulness	1	2	3	4	5
Completeness	1	2	3	4	5
Portability	1	2	3	4	5
Accuracy	1	2	3	4	5
Overall Rating:	**1**	**2**	**3**	**4**	**5**

SECTION 3. COMMENTS

Please comment on any areas where you feel PMC-Sierra could improve the software support and software documentation for this product:

_____

_____

_____

_____

_____

_____

_____

PROPRIETARY AND CONFIDENTIAL TO PMC-SIERRA, INC., AND FOR ITS CUSTOMERS' INTERNAL USE

CONTACTING PMC-SIERRA, INC.

PMC-Sierra, Inc.
105-8555 Baxter Place Burnaby, BC
Canada V5A 4V7

Tel: (604) 415-6000

Fax: (604) 415-6200

Document Information: document@pmc-sierra.com
Corporate Information: info@pmc-sierra.com
Application Information: apps@pmc-sierra.com
Web Site: http://www.pmc-sierra.com

None of the information contained in this document constitutes an express or implied warranty by PMC-Sierra, Inc. as to the sufficiency, fitness or suitability for a particular purpose of any such information or the fitness, or suitability for a particular purpose, merchantability, performance, compatibility with other parts or systems, of any of the products of PMC-Sierra, Inc., or any portion thereof, referred to in this document. PMC-Sierra, Inc. expressly disclaims all representations and warranties of any kind regarding the contents or use of the information, including, but not limited to, express and implied warranties of accuracy, completeness, merchantability, fitness for a particular use, or non-infringement.

In no event will PMC-Sierra, Inc. be liable for any direct, indirect, special, incidental or consequential damages, including, but not limited to, lost profits, lost business or lost data resulting from any use of or reliance upon the information, whether or not PMC-Sierra, Inc. has been advised of the possibility of such damage.

© 1999 PMC-Sierra, Inc.

PM-981296 (R1)