PM5351

S/UNI-TETRA

SOFTWARE DRIVER FOR THE S/UNI-TETRA

PRELIMINARY

ISSUE 1

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

CONTENTS

1 OVERVIEW

1.1 Scope

A software driver for the S/UNI-TETRA (PM5351) 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-TETRA via this API is presented.

The driver is built and tested on the S/UNI-QUAD reference design[1] with a S/UNI-TETRA replacing the S/UNI-QUAD chip. 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-TETRA product, or to incorporate the driver into 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-TETRA drivers for their own systems. It also allows use of the reference design as an evaluation/development vehicle, to gain familiarity with the S/UNI-TETRA 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.

2 SOFTWARE DRIVER FEATURES

The following features are provided by the software driver:

- Driver supports multiple S/UNI-TETRA chips.
- Driver abstracts each channel and chip into a logical device to provide access to these by device ID.
- Driver provides a device data block which 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-TETRA to reduce the coding effort. The defines are parsed from the datasheet and provide a means to access features of the S/UNI-TETRA 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 dependant 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.

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

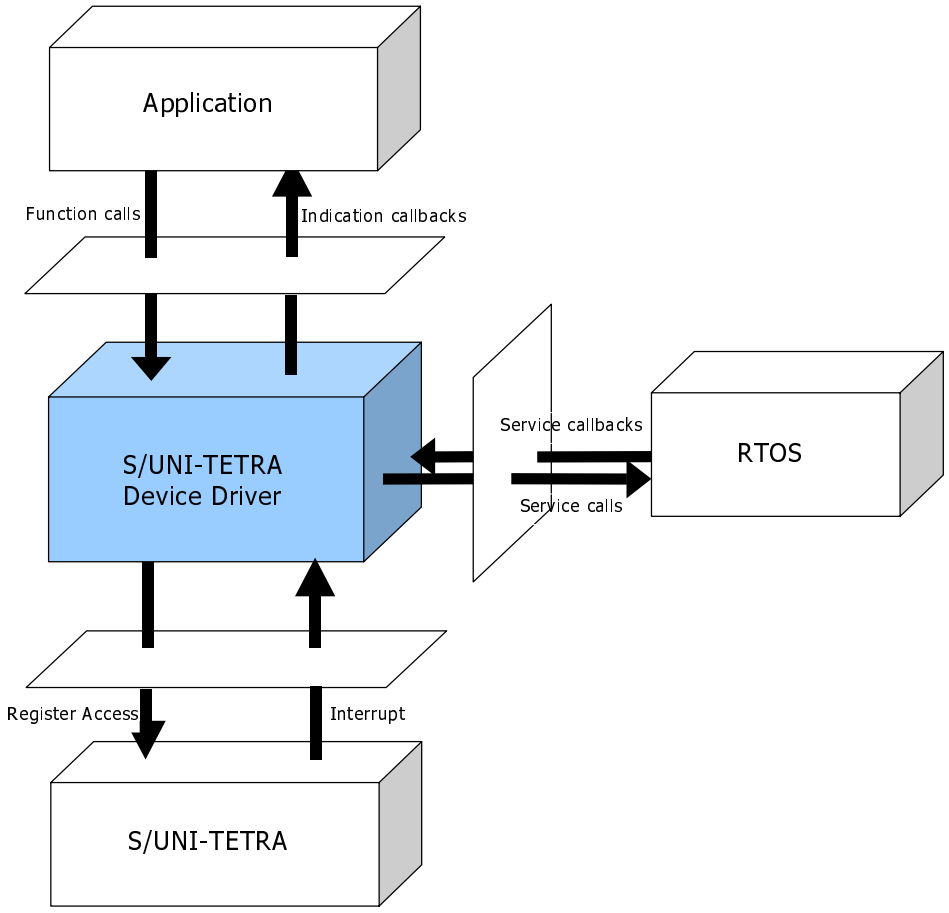
3 APPLICATION PROGRAMMER'S INTERFACE

3.1 Software Architecture

The S/UNI-TETRA 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-TETRA hardware. The arrows show the direction of the function call.

Figure 1. Interfaces of the S/UNI-TETRA Driver



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-TETRA. The function prototypes are shown in section 3.5.

The S/UNI-TETRA has four independent channels that can be seperately controlled and some registers that globally configure the operation of the chip. For this reason the driver has separate data structures associated with each channel and a separate data structure associated with the chip. Each of these data structures provides data for a separately manageable device. There are four channel devices and one chip device per S/UNI-TETRA. The application must identify the device when making a driver function call via a device ID.

When an event occurs within the S/UNI-TETRA the driver must notify the user application that the event occurred. This is done via an Indication function callback with the device identifier 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 the S/UNI-TETRA 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 written such that there is no contention among the application task and the RTOS task for access to a S/UNI-TETRA register or a field within the device data block.

In a multi-tasking environment, where multiple applications may want to access the 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 queing 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-TETRA Hardware Interface

The S/UNI-TETRA hardware interfaces to the driver via register access and via an interrupt pin. There is no DMA interface to the S/UNI-TETRA.

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 "tetra_p.h" and "tetra_p.c".

The registers and bit descriptions of the S/UNI-TETRA have been parsed from the datasheet[2] and placed in the header file "tetra_r.h". Further datasheet related defines are provided in the file "tetra_d.h".

The files "tetra.h" and "tetra.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-TETRA

This software driver provides a framework for users to integrate the S/UNI-TETRA into their own systems. It provides an API of routines that all users of the S/UNI-TETRA would require. Additional features of the S/UNI-TETRA, such as access to overhead, diagnostics or configuration for POS 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.

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 of the third channel device.

```
U8 Value;

DeviceId = DEVICE_ID_CHAN3;

/*modify the transmit path signal label */
tetraWriteRegister(DeviceId,REG_48_TPOP_Path_Signal_Label,0xCF);

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

3.3.2 Power-on Initialization, Self Test and Activation

A user will typically modify the “tetraEntryPoint(…)” 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, init and activate the S/UNI-TETRA 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-TETRA */
tetraEntryPoint();

/* power on self test : configure a diagnostic loopback on channel 4 */
DeviceId = DEVICE_ID_CHAN4;
pDevice = tetraGetDevice(DeviceId);
pDevice->InitVector = NULL;
pDevice->ActivateVector = NULL;
tetraReset(DeviceId);
tetraWriteRegister(DeviceId, REG_07_Channel_Control, BIT_07_SDLE);
tetraInit(DeviceId);
tetraActivate(DeviceId);

/* finished POST so configure all channels with defaults */
DeviceId = DEVICE_ID_CHIP;
pDevice = tetraGetDevice(DeviceId);
pDevice->InitVector = NULL;
pDevice->ActivateVector = NULL;
```

```
tetraReset(DeviceId);
for ( I= FIRST_CHAN_DEVICE_ID; I <= LAST_CHAN_DEVICE_ID; I++) {
    pDevice = tetraGetDevice(I);
    pDevice->InitVector = NULL;
    pDevice->ActivateVector = NULL;
    tetraReset(I);
    tetraInit(I);
    tetraActivate(I);
}
tetraInit(DeviceId);
tetraActivate(DeviceId);
```

3.3.3 Event Notification

Finally a user may need to be notified of an event within the S/UNI-TETRA. This occurs through the driver function “tetraDispatchEvent(..)”. 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 */
tetraReadRegister(DeviceId,REG_33_RPOP_Interrupt_Enable,&Value);

/* modify the interrupt enable bit */
tetraWriteRegister(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 “tetraDispatchEvent(…)” function which is shown below:

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

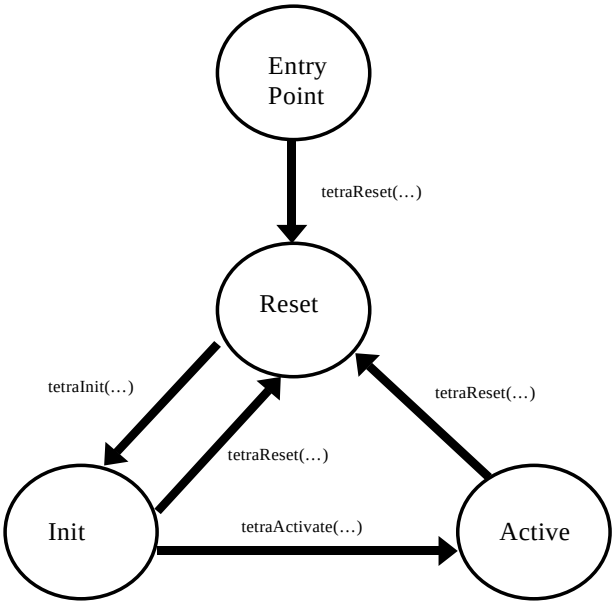
In this driver the event is simply written to the console, but the user would need to port this function to interface to the user’s application, via a task communication mechanism provided by the RTOS. In this driver all interrupt events are assumed to have equal priority and are written to the console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

3.4 Data Structures

3.4.1 tTetraState

The driver maintains one chip device and four channel devices per S/UNI-TETRA. 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.

Figure 2. State Diagram of a Device



The states of a device are defined as follows:

tetraENTRY_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.

tetraRESET: 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.

tetraINIT: The device data block (DDB or CDDB) holds the initialization of the device and and is passed into the tetraInit(…) 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.)

tetraACTIVE: 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 tetraActivate(…) to place the device in the active state. In the active state the S/UNI-TETRA would be able to activate 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 tetraDDB

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

Member	Description
`u32 BaseAddress`	Address of the S/UNI-TETRA in memory space
`tTetraState State`	State of this chip device
`tTetraRegisterValue* InitVector`	Specifies initialization of chip device. Assigned a NULL value for the chip defaults.
`tTetraRegisterValue* ActivateVector`	Specifies interrupt enables for chip device. Assigned a NULL value for chip defaults (all interrupts disabled).
`int DeviceId[SUNI_TETRA_NUMBER_CHANNELS]`	Specifies the channel device IDs belonging to this chip.

u8 CspiIntEn_0C	Interrupt Enable bits of chip device.
u8 CspiIntMask_0C	Interrupt Mask bits to check for active interrupt status.

3.4.3 tetraCDDB

The device context for a S/UNI-TETRA channel is provided in the device data block which has the parameters shown below. This structure is allocated within the function call tetraEntryPoint(…).

Member	Description
u32 BaseAddress	Address of the S/UNI-TETRA channel in memory space. Each channel is offset by 0x100.
tTetraState State	State of this device
tTetraRegisterValue* InitVector	Specifies initialization of channel device. Assigned a NULL value for the chip defaults.
tTetraRegisterValue* ActivateVector	Specifies interrupt enables for channel device. Assigned a NULL value for defaults (all interrupts disabled).
u8 XXXXIntEn_RR	There are 16 interrupt enable registers associated with a channel device. “XXXX” represents the hardware block and “RR” represents the register number.
u8 XXXXIntMask_RR	There are 16 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 channel device. Here “XXXX” represents the 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 tetraEntryPoint

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

3.5.2 tetraExitPoint

Description:	This function does the reverse of the TetraEntryPoint function. It places all devices in the reset state and then deallocates all device resources.
Function Prototype:	**`void tetraExitPoint(void)`**
Function Parameters:	**none**
Return Value:	**none**

Additional Notes: This function is in the porting file “tetra_p.c” because it needs to be modified for the users environment.

3.5.3 tetraReset

Description: This function performs a software reset of a S/UNI-TETRA device and all of its associated channels.

Function Prototype: **`tTetraStatus tetraReset(int DeviceId)`**

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

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

`tetraFAILURE` – invalid device identifier specified.

Additional Notes: none

3.5.4 tetraInit

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

Function Prototype: **`tTetraStatus tetraInit(int DeviceId)`**

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

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

`tetraFAILURE` – invalid device ID

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

3.5.5 tetraActivate

Description: This function places the device in an active state by enabling device interrupts, checking the PHY address and enabling the PHY on the utopia bus.

Function Prototype: **`tTetraStatus tetraActivate(int DeviceId)`**

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

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

`tetraFAILURE` – invalid device ID

Additional Notes:

3.5.6 tetraIsr

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

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

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

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

`tetraFAILURE` – 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 tetraEnableInterrupts

Description: This function enables or re-enables interrupts for the device.

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

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

Return Value: **none**

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

3.5.8 tetraDisableInterrupts

Description: This function disables interrupts for the device.

Function Prototype: **`void tetraDisableInterrupts(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 tetraStatistics

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

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

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

Return Value: none

Additional Notes:

3.5.10 tetraClearCounters

Description: This function clears the accumulated counter values.

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

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

Return Value: none

Additional Notes:

3.5.11 tetraReadRegister

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

Function Prototype: **`bool tetraReadRegister(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 tetraReadSstb

Description: This function reads the indirect S/UNI-TETRA SSTB register associated with the device.

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

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

offset – specifies the register number of the indirect SSTB register 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.

3.5.13 tetraReadSptb

Description: This function reads the indirect S/UNI-TETRA SPTB register associated with the device.

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

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

offset – specifies the register number of the SPTB register 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.

3.5.14 tetraWriteRegister

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

Function Prototype:
```
bool tetraWriteRegister(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 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 tetraWriteSstb

Description: This function writes the indirect S/UNI-TETRA SSTB register associated with the device.

Function Prototype:

```
bool tetraWriteSstb(int DeviceId,
int offset,U8 value)
```

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

offset – specifies the register number of the indirect SSTB register from 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.

3.5.16 tetraWriteSptb

Description: This function writes the indirect S/UNI-TETRA SPTB register associated with the device.

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

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

`offset` – specifies the register number of the indirect SPTB register from 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. **`tetraIsr`**)

`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 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. **tetraStatistics**)

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-TETRA 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 tetraWrite(c, regnum, value) (*(U8*)( c->BaseAddress + (regnum) ) = (value))

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

4 APPENDIX A. SOURCE CODE

Please contact PMC-Sierra to obtain the source code.

5 REFERENCES

[1] PMC-980932, S/UNI-QUAD Reference Design, PMC-Sierra, Issue 1, September 1998.
[2] PMC-971240, S/UNI-TETRA Datasheet, PMC-Sierra, Issue 4, September 1998.

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

PM-981217 (R1)