Platform Abstraction Layer (PAL) Specification

Version 4.2.01

Published by

S2 Technologies, Inc.
1012 2nd Street
Encinitas, CA 92024, USA

The information in this document is subject to change without notice.
Copyright © 2001 – 2010 S2 Technologies, Inc. All rights reserved.

S2 Technologies, the S2 Technologies logo, STRIDE, and the STRIDE logo are trademarks of S2 Technologies, Inc. Microsoft, Windows, and Visual C++ are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. All other trademarks are the property of their respective owners.

Content

1. About This Guide 4
1.1. Purpose 4
1.2. Document Conventions 5
1.3. Standard Data Types 5
1.4. Standard Defines 5
1.5. Hungarian Notation for Variables 5
1.6. Naming Conventions 6
1.7. Grammar 7
1.8. Terms 8
1.9. Related Documents 8
2. Getting Started 9
2.1. What is PAL? 9
2.2. PAL Concepts 9
2.2.1. Function Registration 9
2.2.2. Task Synchronization / Event Notification 9
2.2.3. Protection using Mutex 10
2.2.4. Timer Administration 10
2.2.5. Memory Allocation 10
2.2.6. Transport Services 11
3. PAL Organization 12
3.1. Introduction 12
3.2. PAL Services 12
3.2.1. PAL Operating System Services 12
3.2.2. PAL Input/Output Services 14
3.3. Target Installation Files 15
4. PAL APIs 16
4.1. Introduction 16
4.2. PAL OS Services 16
4.2.1. Synchronization 19
4.2.1.1. palCreateNID() 21
4.2.1.2. palDeleteNID() 23
4.2.1.3. palCreateRFCProxyNID() 24
4.2.1.4. palDeleteRFCProxyNID() 26
4.2.1.5. palWait() 27
4.2.1.6. palNotify() 30
4.2.1.7. palGetThreadId() 32
4.2.1.8. palGetProcessId() 33

4.2.1.9. palSleep() 34
4.2.2. Timers 35
4.2.2.1. palCreateTimer() 36
4.2.2.2. palDeleteTimer() 38
4.2.2.3. palStartTimer() 39
4.2.2.4. palStopTimer() 41
4.2.2.5. palGetTime() 42
4.2.3. Protection using Mutex 43
4.2.3.1. palMutexInit() 44
4.2.3.2. palMutexDestroy() 45
4.2.3.3. palMutexLock() 46
4.2.3.4. palMutexUnlock() 47
4.2.4. Memory Management 48
4.2.4.1. palMemAlloc() 50
4.2.4.2. palMemFree() 51
4.2.4.3. palMemSegmentOpen() 52
4.2.4.4. palMemSegmentClose() 56
4.2.5. Logging 58
4.2.5.1. palLog() 59
4.3. PAL IO Services 61
4.3.1. Function Registration 61
4.3.2. Transmit Data 62
Transmit Data Sequence 62
4.3.2.1. palOutPndReg() 63
4.3.2.2. palOutRdyReg() 65
4.3.2.3. palOut() 67
4.3.3. Receive Data 68
Receive Data Sequence 68
4.3.3.1. palInReg() 69
5. Advanced PAL Usage 71
5.1. Using RFC Proxy Routines 71
5.2. Using the PAL without the RTOS/Scheduler 71
5.3. Calling from an ISR 72
6. S2 SLAP Package 73
6.1. Overview 73
6.1.1. SLAP Frame 73
6.1.2. SLAP Frame Header Format 73
6.1.3. Data Stuffing Characters 74

 Copyright © 2001 – 2010 S2 Technologies, Inc.

6.1.4. Examples of Data Stuffing *75*
6.2. SLAP Services 76
6.3. s2slapTxMsgFormat 77
6.4. s2slapRxMsgExtract 79
6.5. s2slapGetDataReadyCb 81
Appendix A: pal.h 82
Appendix B: s2Slap.h 87

1. About This Guide

1.1. Purpose

This document provides all the information necessary to implement a set of services required by the STRIDE Runtime. This guide specifies the requirements for the PAL Operating System, PAL Input/Output, and Host Transport services.

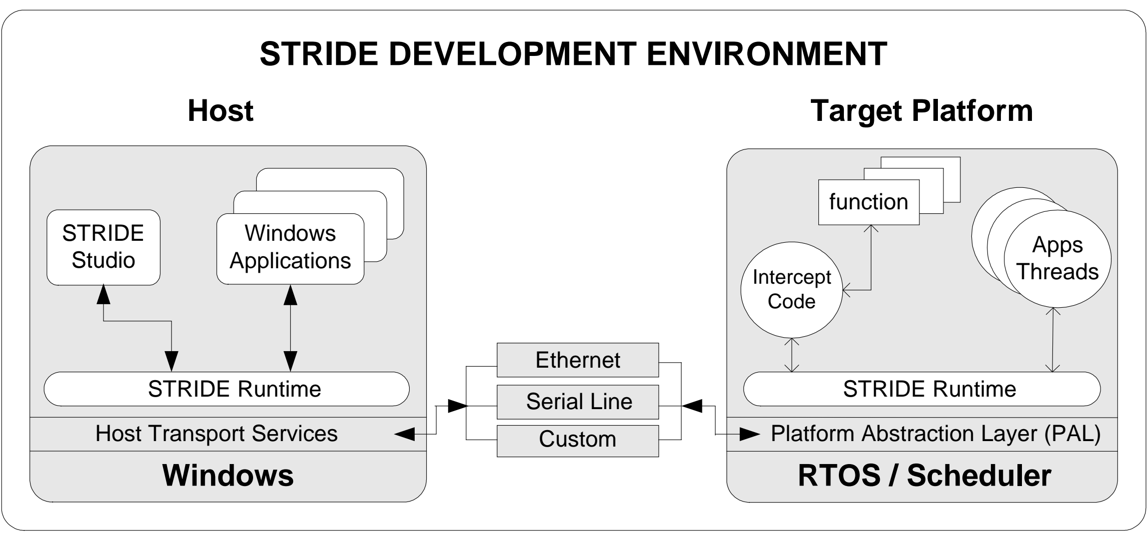
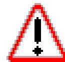


Figure 1. STRIDE Embedded Software Verification Test Platform

 Copyright © 2001 – 2010 S2 Technologies, Inc.

1.2. Document Conventions

This document uses the following conventions:

…	Indicates time passing, activity
	Indicates the developer should take special care to avoid errors
	Indicates additional information that could affect performance
	Indicates interface through use of messaging

1.3. Standard Data Types

The following standard data types, shown in Figure 2, are provided to help self-document the interfaces. The sizes of these types are based on standard Microsoft® Windows® 32-bit platform definitions.

```
typedef char            palCHAR;
typedef unsigned char   palBYTE;
typedef short           palSHORT;
typedef unsigned short  palWORD;
typedef long            palLONG;
typedef unsigned long   palDWORD;
typedef unsigned char   palBOOL;
```

Figure 2. Standard Data Types

1.4. Standard Defines

The PAL also defines TRUE, FALSE and NULL based on standard definitions.

```
#define palFALSE      0
#define palTRUE       1
#define palNULL       0
```

Figure 3. Standard Defines

1.5. Hungarian Notation for Variables

The naming convention for variables used by the PAL API follows a modified version of the Hungarian notation. Each variable name begins with one or more lowercase characters identifying the type of the variable.

Of special note are the enumeration type and the more general typedef:

The "_e" notation indicates that an enum typedef is being used.

The “_t” indicates a general typedef.

Variables declared as an enumeration or a general typedef uses “e” or “t” in the prefix.

Table 1. Hungarian Notation

Prefix	Meaning	Example	
c	char	palCHAR	cMyChar;
y	unsigned char	palBYTE	yMyByte;
n	short	palSHORT	nMyShort;
w	unsigned short	palWORD	wMyWord;
l	long	palLONG	lMyLong;
dw	unsigned long	palDWORD	dMyDWord;
b	Boolean	palBOOL	bMyBool;
e	enumeration	<Name>_e	eMyEnum;
t	typedef	<Name>_t	tMyTypedef;
p	pointer	palBOOL	*pbMyPtrBool;
sz	zero terminated string	palCHAR	*szMyString;

1.6. Naming Conventions

All **public** header files, prototypes, data types, constants, and variables use the **component tag** (e.g., lower case “**pal**”) as a prefix. The following naming conventions are used:

Item	Convention	Example
Files	<tag><name>**.h | .c**	pal.h
Prototypes	<tag><Name>(…)	palMyFunction(..)
Typedefs	<tag><Name>**_t**	palMyType_t
Constants	<tag><NAME>{_<NAME>}	palMY_CONSTANT
Enumeration	<tag><Name>**_e**	palMyEnum_e
Enumerator	<tag><NAME>{_<NAME>}	palMY_ENUMERATOR

All **private** files also use the component tag as well as the module name. An additional underscore (“_”) is inserted in front of the component tag as a prefix for private prototypes and variables with global scope. Static variables defined within a module, as well as local constants and typedefs, do not follow any specific convention.

Copyright © 2001 – 2010 S2 Technologies, Inc.

Item	Convention	Example
Files	<tag><module-name>**.h**	palmod.h
	<tag><module-name>**.c**	palmod.c
Prototypes	_<tag><Module-name>_<Name>(…)	_palMod_Func(..)
Variable	_<tag><Module-name>_<Name>	_palMod_Variable
Typedefs	<Module-name>_<Name>**_t**	Mod_Type_t
Constants	<Module-name>_<NAME>{_<NAME>}	MOD_CONSTANT
Enumeration	<Module-name>_<Name>**_e**	Mod_Enum_e
Enumerator	<Module-name>_<NAME>{_<NAME>}	MOD_ENUMERATOR

1.7. Grammar

Item	Grammar
<name>	<alpha-char> {<alpha-char>}
<alpha-char>	a | b | c | .. | z
<NAME>	<ALPHA-CHAR> {<ALPHA-CHAR>}
<ALPHA-CHAR>	A | B | C | .. | Z
<Name>	<ALPHA-CHAR> {<string>}
<string>	<Alpha-Char> {<Alpha-Char>}
<Alpha-Char>	<ALPHA-CHAR> | <alpha-char>
<Module-name>	<ALPHA-CHAR> {<string>}
<module-name>	<alpha-char> {<alpha-char>}
<tag>	<alpha-char> {<alpha-char>}

Copyright © 2001 – 2010 S2 Technologies, Inc.

1.8. Terms

I-block	STRIDE Communication Model (SCM) term for a packet of data transferred between platforms
message	A communication mechanism between two threads
module	A file containing one (1) or more functions
NID	Notification Identifier
pool memory	Memory allocated from a common pool used by application threads
private memory	Non-pool memory that is owned by a sending application thread
process	Implies a separate address space which typically does not apply to a task or thread
proxy	Software that uses an interface to connect a user to a remote device
public interface	Exposed to another component/unit
RFC	Remote function call
sender	The originator of a message
stub	Temporary code written to replace a unit that has yet to be written or is otherwise unavailable
task	Often used interchangeably with “thread”
thread	An independent entity running under the control of an Operating System
Transport	A communication link that connects the target with the host

1.9. Related Documents

The following publications are also available through STRIDE Online Help:

STRIDE Host Runtime Transport Specification

STRIDE Runtime Developer’s Guide

STRIDE Communication Language Reference Guide

Copyright © 2001 – 2010 S2 Technologies, Inc.

2. Getting Started

2.1. What is PAL?

The PAL, or Platform Abstraction Layer, provides a consistent interface for the STRIDE Runtime regardless of the operating system or data transport used. This interface layer is necessary given the broad variety of operating systems and data transports that exist within embedded systems today.

A small set of functions, written according to the PAL specification, provides a virtual link between your operating system and platform transport mechanism to the STRIDE Runtime. Through the Platform Abstraction Layer, the STRIDE Runtime becomes independent of any specific operating system or transport. The PAL is designed to use standard concepts and services present in almost all operating systems and transport mechanisms. To complete the PAL, you'll need to be familiar with concepts such as event signaling, scheduling, timers, critical section protection, memory allocation and data transfers.

The "pal.h" header file provided with the STRIDE installation contains all the function prototypes necessary for writing the PAL. The pal.h header file is provided in *Appendix A* beginning on page 82. For more information on the STRIDE Runtime, see the *STRIDE Runtime Developer's Guide* PDF .

2.2. PAL Concepts

Use this section to familiarize yourself with the basic concepts of the PAL. Subsequent sections build on these concepts providing you with the PAL organization and API.

2.2.1. Function Registration

Typically, the STRIDE Runtime calls PAL functions; however, there are some functions in the STRIDE Runtime that need to be called by the PAL. To eliminate PAL dependencies on the STRIDE Runtime, these functions are accessed through a function registration process initiated by the STRIDE Runtime at startup. You write the registration routine called by the Runtime. This registration routine passes in the address of the STRIDE Runtime function to be registered as the input parameter, and stores the address of the STRIDE Runtime function in your own function variable, which is actually a pointer to a function. You can then call the registered function using your function pointer variable.

The registered STRIDE Runtime functions allow you to complete such tasks as delivering a received I-block to the Runtime, checking the number of I-blocks the Runtime has ready to send out, or signalling the Runtime that your transport is ready for the next I-block. It is not necessary for the PAL to know the details of these Runtime functions.

2.2.2. Task Synchronization / Event Notification

Unique information is required by an operating system to notify a thread of a pending event. This information can be a simple index into a table, an address to a thread control

Copyright © 2001 – 2010 S2 Technologies, Inc.

block, the address of a semaphore or one of a number of other implementations. Although each implementation may be different, a unique notifier is necessary for each thread. The STRIDE Runtime calls this unique information a Notification Identifier (NID). A thread uses a NID when waiting for a STRIDE event, and the STRIDE Runtime uses the same NID to notify the thread of a pending STRIDE event.

An additional piece of information included with the notify routine is a box ID, which is the ID of the mailbox where the message is delivered.

Support for sharing of the synchronization object among multiple applications should be ensured in case of multi-process target is enabled.

2.2.3. Protection using Mutex

The STRIDE Runtime needs to protect critical and shared data structures from multiple, simultaneous accesses by multiple threads and, in case of multi-process target is enabled, by multiple applications. The PAL requires implementing protection using Mutexes that can be created and, in case of multi-process target is enabled, used with a unique name by any STRIDE Runtime module or PAL itself. The STRIDE Runtime guarantees that calls to **palMutexLock()** will not be nested.

2.2.4. Timer Administration

The STRIDE Runtime requires that at least one timer be available. When a timer is created, a callback is registered so that the STRIDE Runtime can be notified of timer expirations. A user parameter, provided when the timer is created, is passed to the callback when it gets called. If several timers share a callback routine, this user parameter can be used to identify which timer expired. The STRIDE Runtime can also stop, start and delete timers.

Because the STRIDE Runtime also has the need to timestamp trace log data, a routine that returns the system time, **palGetTime(),** is also part of the PAL. This routine is not associated with timers.

2.2.5. Memory Allocation

The STRIDE Runtime dynamically allocates memory for messages and trace log storage. The Runtime uses **palMemAlloc()** and **palMemFree()** to allocate memory dynamically and then return it to the system.

The Runtime uses **palMemSegmentOpen()** and **palMemSegmentClose()** to create, open and close memory segments. In case of single process target, these routines can simply allocate dynamic memory as in normal memory allocation routines.

To support multi-process target, the STRIDE Runtime requires dynamic and configurable memory allocated through **palMemAlloc()** and static internal memory allocated directly through **palMemSegmentOpen()** to be shared memory.

In case of multi-process target, **palMemAlloc()** and **palMemFree()** routines can simply call the STRIDE Runtime's Memory Management module *srMem*, which is in turn dependent on **palMemSegmentOpen()** and **palMemSegmentClose()** routines.

 Copyright © 2001 – 2010 S2 Technologies, Inc.

2.2.6. Transport Services

The PAL transport routines are needed to transfer I-blocks (STRIDE data packets) into and out of the STRIDE Runtime on your target platform. These routines handle the buffering and transferring of data to and from your transport mechanism. The PAL also contains registration routines that allow for STRIDE Runtime routines to be called by the PAL.

The **palOut()** routine enables the STRIDE Runtime to transfer I-blocks to your transport. The STRIDE Runtime calls the **palOut()** routine whenever it needs to send out an I-block.

Your transport calls the routine registered with the **palOutRdyReg()** routine when the transport is ready for the next I-block to be transmitted. The STRIDE Runtime will not call the **palOut()** routine until you call the registered function. In this way, you can control the flow of transmitted I-blocks.

When a complete I-block has been received by your transport, the routine registered with the **palInReg()** routine should be called to put the received I-block into the STRIDE Runtime.

In some cases it is useful to know when the transmit path is not being requested. Your transport mechanism can check the number of I-blocks the STRIDE Runtime has ready for transmission by calling the routine registered with the **palOutPndReg()** routine.

For further information on transports, refer to the *STRIDE Host Runtime Transport Specification* PDF .

3. PAL Organization

3.1. Introduction

This section explains the two sets of PAL services that support the STRIDE Runtime, as well as the files used to implement these services on your target.

3.2. PAL Services

The PAL services include the following:

Operating System (OS) Services

Input/Output (IO) Services

3.2.1. PAL Operating System Services

The PAL Operating System (OS) Services are the routines that enable the STRIDE Runtime to work with the operating system on your target platform. In order to write your PAL OS services you must have detailed knowledge of how your operating system handles thread synchronization, timers, mutexes, dynamic memory, shared memory, and notification. The PAL OS services make the features of your operating system available to the STRIDE Runtime.

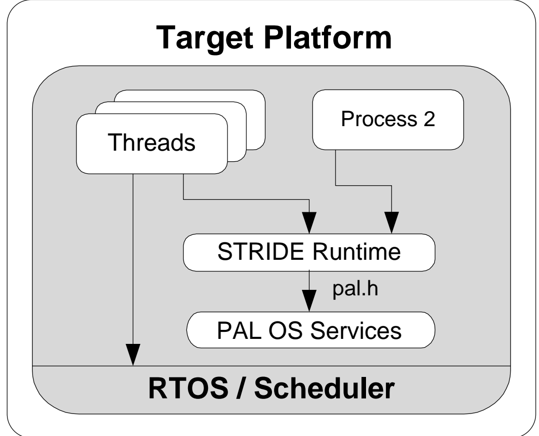


Figure 4. PAL OS Services

Copyright © 2001 – 2010 S2 Technologies, Inc.

The PAL OS Services shown in Table 2 are defined in the pal.h file. These functions need to be fully implemented.

Table 2. PAL OS Functions

Function Name	Description
Synchronization	
palCreateNID	Create a Notification Identifier (NID)
palDeleteNID	Delete a Notification Identifier (NID)
palCreateRFCProxyNID	Create the RFC Proxy Notification Identifier (NID)
palDeleteRFCProxyNID	Delete the RFC Proxy Notification Identifier (NID)
palWait	Wait for an event
palNotify	Signal a thread that an event is pending
palGetThreadId	Return current thread Id
palGetProcessId	Return current process Id
palSleep	Suspend the execution of the current thread
Timers	
palCreateTimer	Create a timer
palDeleteTimer	Delete a timer
palStartTimer	Start a timer
palStopTimer	Stop a timer
palGetTime	Return system time (e.g., tick count)
Mutex	
palMutexInit	Initialize a mutex object
palMutexDestroy	Destroy a mutex object
palMutexLock	Lock a mutex object
palMutexUnlock	Unlock a mutex object
Memory Management	
palMemAlloc	Allocate a block of dynamic memory
palMemFree	Free a block of dynamic memory
palMemSegmentOpen	Create/open memory segments
palMemSegmentClose	Close memory segments
Logging *(Optional)*	
palLog	Log messages according to log-level and formatted string

Copyright © 2001 – 2010 S2 Technologies, Inc.

3.2.2. PAL Input/Output Services

The PAL Input/Output (IO) Services are the routines that enable the STRIDE Runtime to work with different data transport mechanisms. These routines enable your transport to send and receive data between the host and the target.

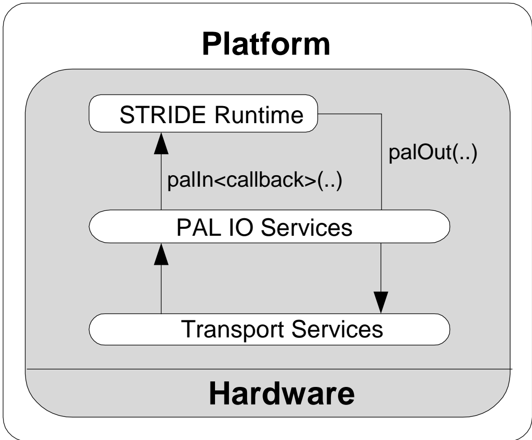


Figure 5. PAL IO Services

The IO services have also been defined to allow the target platform to control how memory is managed and the rate of data exchange. See *Appendix* A beginning on page 82 for definitions of the header file functions listed in Table 3.

Table 3. PAL IO Functions

Function Name	Description
Transmit	
palOutPndReg	Query the current output queue of the Runtime
palOutRdyReg	Identify the transport as ready to receive data
palOut	Send data to the host platform
Receive	
palInReg	Data extracted from the transport and identified for the STRIDE Runtime

Copyright © 2001 – 2010 S2 Technologies, Inc.

3.3. Target Installation Files

The following is a list of all the files necessary to implement and test a fully functional Platform Abstraction Layer (PAL) for the target platform.

PAL and SLAP files are installed along with STRIDE Runtime files as part of STRIDE SDK into <STRIDE_DIR>\SDK\<Platform>\src folder.

File Name	Description
pal.h	Prototypes for Platform Abstraction Layer (PAL)
palIO.c	Template implementation of the PAL IO services
palOS.c	Template implementation of the PAL OS services
s2Slap.h	Simplified Link Application Protocol Prototypes
s2Slap.c	Simplified Link Application Protocol Implementation

Copyright © 2001 – 2008 S2 Technologies, Inc.

4. PAL APIs

4.1. Introduction

This section provides the parameters and return values of each PAL API function. The PAL API is organized into two main groups: PAL OS Services (beginning below), and PAL IO Services (beginning on on page 61).

4.2. PAL OS Services

The PAL OS Services are consisted of a set of functions required by the STRIDE Runtime to operate on your platform. The Runtime has been designed to be independent of any specific Real-Time Operating System (RTOS). The PAL defines the required services and abstracts the details of any implementation. The palOS.c source file can be used as a starting point as it contains template code for all of the OS functions. Full implementation of the following PAL OS functions is required except when stated *optional*:

Function Name	Description
Synchronization	**page 19**
palCreateNID	Create a Notification Identifier (NID)
palDeleteNID	Delete a Notification Identifier (NID)
palCreateRFCProxyNID	Create the RFC Proxy Notification Identifier (NID)
palDeleteRFCProxyNID	Delete the RFC Proxy Notification Identifier (NID)
palWait	Wait for an event
palNotify	Signal a thread that an event is pending
palGetThreadId	Return current thread Id
palGetProcessId	Return current process Id
palSleep	Suspend the execution of the current thread
Timers	**page 35**
palCreateTimer	Create a timer
palDeleteTimer	Delete a timer
palStartTimer	Start a timer
palStopTimer	Stop a timer
palGetTime	Return system time (e.g., tick count)
Mutex	**page 43**
palMutexInit	Initialize a mutex object
palMutexDestroy	Destroy a mutex object
palMutexLock	Lock a mutex object

 Copyright © 2001 – 2010 S2 Technologies, Inc.

palMutexUnlock	Unlock a mutex object
Memory Management	**page 48**
palMemAlloc	Allocate a block of dynamic memory
palMemFree	Free a block of dynamic memory
palMemSegmentOpen	Create/open memory segments
palMemSegmentClose	Close memory segments
Logging *(Optional)*	**page 58**
palLog	Log messages according to log-level and formatted string

Note: Generic RTOS service function names are used throughout the implementation examples in this document; they are placeholders that correspond to the real functionality of the target platform being used. ***These need to be replaced with the corresponding function(s) available on your target***. Brief descriptions of these functions are as follows:

Function Name	Description of RTOS Service
rtosEventSet	Set an event
rtosEventWait	Pend on an event
rtosThreadId	Current thread Id
rtosProcessId	Current process Id
rtosSleep	Suspend execution of current thread
rtosMutexInit	Initialize a mutex object
rtosMutexDestroy	Destroy a mutex object
rtosMutexLock	Lock a mutex object
rtosMutexUnlock	Unlock a mutex object
rtosGetTick	Get the system timer tick
rtosCreateThread	Create and initialize a thread in the RTOS
rtosCreateTimer	Create a Timer
rtosStartTimer	Start a Timer
rtosStopTimer	Stop a Timer
rtosMalloc	Allocate dynamic memory block
rtosFree	Free memory block allocated by rtosMalloc
rtosShm_open	Open existing shared memory object
rtosShm_unlink	Close/unlink existing shared memory object

Copyright © 2001 – 2008 S2 Technologies, Inc.

rtosMmap	Map shared memory object
rtosMunmap	Unmap shared memory object
rtosFtruncate	Set size of shared memory object
rtosCloseHandle	Close handle
rtosTransportWrite	Write data to the transport

Copyright © 2001 – 2010 S2 Technologies, Inc.

4.2.1. Synchronization

The PAL synchronization services are required by the Runtime, Stub and Proxy code for notification and scheduling. The synchronization routines include:

palCreateNID()

palDeleteNID()

palCreateRFCProxyNID()

palDeleteRFCProxyNID()

palWait()

palNotify()

palGetThreadId()

palGetProcessId()

palSleep()

When supporting a full-featured RTOS, a NID can be the identifier of a binary semaphore, a set of operating system mailboxes, or a set of software signals. The NID must uniquely identify a resource that can be used to cause a thread to wait and notify a waiting thread that it no longer needs to wait. Your system will require at least one NID, but may require more, depending on the use of Stub and Proxy code.

The STRIDE Runtime runs primarily in the context of the STRIDE Runtime Thread. The STRIDE Runtime Thread uses **palCreateNID()** to allocate one NID for its synchronization use. STRIDE Stub Threads, launched by the application when the application contains Stub code, will also use one NID per thread, acquired with a call to **palCreateNID().** Neither the STRIDE Runtime Thread nor Stub Threads will delete their NIDs.

When application threads use RFC Proxies, one NID is required for each outstanding RFC Proxy function call. These NIDs are acquired with a call to **palCreateRFCProxyNID().** When the remote function call returns, the NID is deleted using **palDeleteRFCProxyNID().** Because RFC Proxy NIDs can be created and deleted, it may make sense in the PAL implementation to allow them to be reused. Also, depending on the synchronization primitives being used for NIDs and Proxy NIDs, it may be possible to have **palCreateRFCProxyNID()** and **palDeleteRFCProxyNID()** simply call **palCreateNID()** and **palDeleteNID(),** respectively.

Proxy NIDs may require special attention in your PAL implementation. For more about how to use Proxy NIDs, see *Using RFC Proxy Routines*, page 71.

The STRIDE Runtime Thread calls **palWait()** to wait for a notification of a received message or expiration of its PAL timer. The Runtime thread is woken up by a call to **palNotify().** The STRIDE Runtime thread also calls **palNotify()** to wake up other threads when they receive events, using their respective NIDs to identify them.

Copyright © 2001 – 2008 S2 Technologies, Inc.

The STRIDE Runtime thread will not call **palDeleteNID().** Only Proxy functions will delete NIDs, using **palDeleteRFCProxyNID().**

The **palNotify()** routine has a box ID as an input parameter. The STRIDE Runtime will pass in a box ID when calling **palNotify(),** but this parameter is not used to notify the Runtime or the Stub threads. The box ID indicates which mailbox has events pending, which is useful for applications using the STRIDE Messaging Model. For **palNotify(),** this parameter can be used to generate different types of OS events, depending on which box is being notified.

Copyright © 2001 – 2010 S2 Technologies, Inc.

4.2.1.1. palCreateNID()

Create Notification Identifier

Prototype

```
palBOOL palCreateNID( palDWORD *pdwNID );
```

Description

The **palCreateNID()** routine is used by the STRIDE Runtime thread to create a Notification Identifier (NID). The NID is a value that is used to identify a synchronization object, such as a semaphore, event group, or mail queue.

Parameters	Type	Description
[out] pdwNID	palDWORD *	The created Notification Identifier (NID)

Return Value	Type	Description
	palBOOL	palTRUE if successfully created, palFALSE otherwise.

Example

```
#include "pal.h"

#define PAL_MAX_THREADS 8
#define PAL_SYNCH_ID "s2sPAL{F8227755-21FE-4cef-A3DD-3702DB699C16}"

typedef struct{
    void*    pSynchObj;
    palDWORD dwNID;
} ThreadCB_t;
static ThreadCB_t tThreadCB[PAL_MAX_THREADS] = {{palNULL, 0}};

palBOOL palCreateNID( palDWORD *pdwNID )
{
    palBOOL bSuccess;
    palWORD wIdx;

    /* Find an empty spot in the NID table */
    for (wIdx = 0; wIdx < PAL_MAX_THREADS; wIdx++)
    {
        if(tThreadCB[wIdx].dwNID == 0)
            break;
```

```
    }

    /* Check to see if there was an available NID */
    if(wIdx < PAL_MAX_THREADS)
    {
        palCHAR szName[128];

        /* Set NID          */
        *pdwNID = palGetThreadId();

        /* Mark NID as used */
        tThreadCB[wIdx].dwNID = *pdwNID;

        /* Any additional initialization for Synchronization Object
           can also be done in this function */
        snprintf(szName, 128, "%s%04x", PAL_SYNCH_ID, *pdwNID);
        bSuccess = palMutexInit(&tThreadCB[wIdx].pSyncObj, szName);
        if (bSuccess == palTRUE)
        {
            bSuccess = palMutexLock(tThreadCB[wIdx].pSyncObj);
        }
    }
    else
    {
        /* There are no NIDs available */
        bSuccess = palFALSE;
    }

    return bSuccess;
}
```


Copyright © 2001 – 2010 S2 Technologies, Inc.

4.2.1.2. palDeleteNID()

Delete Notification Identifier

Prototype

```
palBOOL palDeleteNID( palDWORD  dwNID );
```

Description

The **palDeleteNID()** routine is used to indicate that a NID is no longer in use and its resources should be freed. This routine is not called by the STRIDE Runtime.

Parameters	Type	Description
[in] dwNID	palDWORD	The created Notification Identifier (NID)

Return Value	Type	Description
	palBOOL	palTRUE if successfully deleted, palFALSE otherwise.

Example

```
#include "pal.h"

/* See palCreateNID() example */

palBOOL palDeleteNID( palDWORD dwNID )
{
    palWORD wIdx;

    for (wIdx = 0; wIdx < PAL_MAX_THREADS; wIdx++)
    {
        if (dwNID == tThreadCB[wIdx].dwNID)
        {
            /* Set NID to available in table */
            tThreadCB[wIdx].dwNID = 0;

            /* Any additional clean-up/teardown for
            Synchronization Object can also be done in this function */
            palMutexDestroy(tThreadCB[wIdx].pSyncObj);
            tThreadCB[wIdx].pSyncObj = NULL;

            break;
        }
    }

    return palTRUE;
}
```

Copyright © 2001 – 2008 S2 Technologies, Inc.

4.2.1.3. palCreateRFCProxyNID()

Create RFC Proxy Notification Identifier

Prototype

```
palBOOL palCreateRFCProxyNID( palDWORD *pdwNID );
```

Description

The **palCreateRFCProxyNID()** routine is used by the STRIDE proxy function to create a Notification Identifier (NID). The NID is a value that is used to identify a synchronization object, such as an ID or pointer to a semaphore, event group, or mail queue. Depending on the type of synchronization primitive chosen (see *Synchronization* on page 19) it may be possible to implement **palCreateRFCProxyNID()** by calling palCreateNID().

Parameters	Type	Description
[out] pdwNID	palDWORD *	The created Notification Identifier (NID)

Return Value	Type	Description
	palBOOL	palTRUE if successfully created, palFALSE otherwise.

Example

```
#include "pal.h"

struct{
    rtosEvent_t   Events;  /* RTOS Event Group                  */
    palBOOL       bUsed;   /* Flag set if Event Group is in use */
} NID_Pool[NID_POOL_SIZE];

palBOOL  palCreateRFCProxyNID( palDWORD *pdwNID )
{
    palCHAR yPoolIdx;
    palBOOL  bReturn = palTRUE;

    /* Find the next available event group */
    for( yPoolIdx = 0; yPoolIdx < NID_POOL_SIZE; yPoolIdx++ )
    {
        if(NID_Pool[yPoolIdx].bUsed == palFALSE)
            break;
    }
```


Copyright © 2001 – 2010 S2 Technologies, Inc.

```
    /* return false if all event groups are in use */
    if( yPoolIdx == NID_POOL_SIZE )
        bReturn = palFALSE;
    else
    {
        /* Indicate event group is in use */
        NID_Pool[yPoolIdx].bUsed = palTRUE;
        /* Use the table index as the Notify ID.*/
        *pdwNID = yPoolIdx;
    }
    return bReturn;
}
```

4.2.1.4. palDeleteRFCProxyNID()

Delete RFC Proxy Notification Identifier

Prototype

```
palBOOL palDeleteRFCProxyNID( palDWORD  dwNID );
```

Description

The **palDeleteRFCProxyNID()** routine is called by the STRIDE Runtime when the RFC Proxy NID is no longer in use and its resources can be released.

Parameters	Type	Description
[in] dwNID	palDWORD	The created Notification Identifier (NID)

Return Value	Type	Description
	palBOOL	palTRUE if successfully deleted, palFALSE otherwise.

Example

```
#include "pal.h"

/* See palCreateNID() example */

palBOOL  palDeleteRFCProxyNID( palDWORD dwNID )
{
    palBOOL bReturn = palTRUE;
    /* Set the used flag to false to free up
       event group */
    if( dwNID > NID_POOL_SIZE )
        bReturn = palFALSE;
    else
        NID_Pool[dwNID].bUsed = palFALSE;
    return bReturn;
}
```


Copyright © 2001 – 2010 S2 Technologies, Inc.

4.2.1.5. palWait()

Wait for event

Prototype

```
palBOOL palWait( palDWORD dwNID , palDWORD *pdwEvents );
```

Description

The **palWait()** routine is used by the STRIDE Runtime Thread, Stub Threads, and threads that call RFC Proxies, which wait for events to occur. The calling thread will be suspended until **palNotify()** is called with the NID and events that match the NID and events parameters passed in to **palWait().**

Parameters	Type	Description
[in] dwNID	palDWORD	The created Notification Identifier (NID)
[in/out] pdwEvents	palDWORD *	Events notification where palSTOP_EVENT on stop, Bits 0-7 for mail boxes and the rest custom specified. On input the value should be set to a mask expected events and on output it would contain a mask of the notified events.

Return Value	Type	Description
	palBOOL	palTRUE if successfully received notification, palFALSE otherwise.

Example

```
#include "pal.h"
#include "srcfg.h"

/* See palCreateNID() example */

palBOOL palWait( palDWORD dwNID , palDWORD *pdwEvents )
{
    palBOOL  bSuccess = palFALSE;
    palWORD wIdx;

    /* Identify the slot in the NID table */
    for (wIdx = 0; wIdx < PAL_MAX_THREADS; wIdx++)
    {
        if (dwNID == ppalContext->tThreadCB[wIdx].dwNID)
        {
            *pdwEvents  |= palSTOP_EVENT;
```

```
            bSuccess = palTRUE;
            break;
        }
    }

    /* Wait on synchronization object and requested events */
    while (bSuccess == palTRUE && (ppalContext->tThreadCB[wIdx].dwEvents &
*pdwEvents) == 0)
    {
#if srCFG_MULTI_PROC_TARGET
        palCHAR szName[PAL_MODULE_STR_SIZE];
        void *pObj;

        PAL_GET_SYNCH_NAME(szName, dwNID);
        bSuccess = palMutexInit(&pObj, szName);
        if (bSuccess == palTRUE)
        {
            bSuccess = palMutexLock(pObj);
            palMutexDestroy(pObj);
        }
#else /* NOT srCFG_MULTI_PROC_TARGET */
        bSuccess = palMutexLock(ppalContext->tThreadCB[wIdx].pSyncObj);
#endif /* srCFG_MULTI_PROC_TARGET */
    }

    if (bSuccess == palTRUE)
    {
        palDWORD dwFlags;
        /* Return flags that were signaled */
        dwFlags = ppalContext->tThreadCB[wIdx].dwEvents;
        /* clear only requested flags in task signals */
        ppalContext->tThreadCB[wIdx].dwEvents &= ~(*pdwEvents);
        *pdwEvents = dwFlags;
    }
    else
    {
        palLog(palLOG_LEVEL_ERROR, "palWait failed");
    }

    return bSuccess;
```

 Copyright © 2001 – 2010 S2 Technologies, Inc.

```
}
```

4.2.1.6. palNotify()

Signal event pending

Prototype

```
palBOOL palNotify( palDWORD dwNID, palDWORD dwEvents );
```

Description

The **palNotify()** routine is used to notify a waiting thread that an event is pending. The thread is identified by the unique NID passed in. The event bit mask is used by the STRIDE Runtime to indicate termination of thread and/or which mailbox has pending events and/or any user specified tasks.

Parameters	Type	Description
[in] dwNID	palDWORD	The created Notification Identifier (NID)
[in] dwEvents	palDWORD	Events to notify where palSTOP_EVENT for stop, Bits 0-7 for mail boxes and the rest custom specified

Return Value	Type	Description
	palBOOL	palTRUE if successfully issued notification, palFALSE otherwise.

Example

```
#include "pal.h"
#include "srcfg.h"

/* See palCreateNID() example */

palBOOL  palNotify( palDWORD dwNID, palDWORD dwEvents )
{
    palBOOL  bSuccess = palFALSE;
    palWORD wIdx;

    /* Identify the slot in the NID table */
    for (wIdx = 0; wIdx < PAL_MAX_THREADS; wIdx++)
    {
        if (dwNID == ppalContext->tThreadCB[wIdx].dwNID)
        {
            bSuccess = palTRUE;
            break;
```

 Copyright © 2001 – 2010 S2 Technologies, Inc.

```
        }
    }

    if (bSuccess == palTRUE && (ppalContext->tThreadCB[wIdx].dwEvents &
dwEvents) != dwEvents)
    {
        /* Set the event notification */
        ppalContext->tThreadCB[wIdx].dwEvents |= dwEvents;

        /* Signal synchronization object */
#if srCFG_MULTI_PROC_TARGET
        {
            palCHAR szName[PAL_MODULE_STR_SIZE];
            void *pObj;

            PAL_GET_SYNCH_NAME(szName, dwNID);
            bSuccess = palMutexInit(&pObj, szName);
            if (bSuccess == palTRUE)
            {
                bSuccess = palMutexUnlock(pObj);
                palMutexDestroy(pObj);
            }
        }
#else /* NOT srCFG_MULTI_PROC_TARGET */
        bSuccess = palMutexUnlock(ppalContext->tThreadCB[wIdx].pSyncObj);
#endif /* srCFG_MULTI_PROC_TARGET */
    }

    if (bSuccess != palTRUE)
    {
        palLog(palLOG_LEVEL_ERROR, "palNotify failed");
    }

    return bSuccess;
}
```

4.2.1.7. palGetThreadId()

Return current thread Id

Prototype

```
palDWORD palGetThreadId( void );
```

Description

The **palGetThreadId()** routine is used when the Runtime or STRIDE generated code needs to know the current thread id.

Parameters	Type	Description
None		

Return Value	Type	Description
	palDWORD	Current thread Id

Example

```
#include "pal.h"

palDWORD palGetThreadId( void )
{
    return rtosThreadId();
}
```


Copyright © 2001 – 2010 S2 Technologies, Inc.

4.2.1.8. palGetProcessId()

Return current process Id

Prototype

```
palDWORD palGetProcessId( void );
```

Description

The **palGetProcessId()** routine is used when the Runtime or STRIDE generated code needs to know the current process id.

Parameters	Type	Description
None		

Return Value	Type	Description
	palDWORD	Current process Id

Example

```
#include "pal.h"


palDWORD palGetProcessId( void )
{
    return rtosProcessId();
}
```

4.2.1.9. palSleep()

Suspend the execution of current thread for a given period

Prototype

```
void palSleep( palDWORD dwTimeout );
```

Description

The **palSleep()** routine is used when the Runtime or STRIDE generated code needs to suspend the execution of the current thread.

Parameters	Type	Description
[in] dwTimeout	palDWORD	Duration for sleep in milliseconds.

Return Value	Type	Description
None		

Example

```
#include "pal.h"


void palSleep( palDWORD dwTimeout )
{
    return rtosSleep(dwTimeout);
}
```


Copyright © 2001 – 2010 S2 Technologies, Inc.

4.2.2. Timers

The PAL Timer routines are used by the STRIDE Runtime thread to control the starting and stopping of a periodic timer. Timer routines include:

palCreateTimer()

palDeleteTimer()

palStartTimer()

palStopTimer()

palGetTime()

The STRIDE Runtime thread calls **palCreateTimer()** once at startup. The Runtime thread then calls **palStartTimer()** and **palStopTimer()** to control the starting and stopping of the timer. The STRIDE Runtime creates only one timer. RFC Stub threads and Proxy threads do not use timers.

The user ID value passed in to the **palCreateTimer()** routine is passed back to the callback routine, which runs when the timer expires.

The STRIDE Runtime thread uses the **palGetTime()** routine to get your current system time. This routine is not connected to the use of timers, but simply captures the system time in ticks. The STRIDE Runtime interprets ticks based on the parameters in srcfg.h. The information in srcfg.h defines the time units (seconds/tick, milliseconds/tick, or microseconds/tick) and duration of each tick (1 unit/tick, 5 units/tick…etc.). Refer to the **STRIDE Runtime Developer's Guide** (available through STRIDE Online Help) for more about srcfg.h.

4.2.2.1. palCreateTimer()

Create timer

Prototype

```
typedef void (*palTimerCallback_t) ( void *pvUser );
palBOOL palCreateTimer( palTimerCallback_t  pFuncCb,
                        void               *pvUser,
                        palDWORD           *pdwTimerId );
```

Description

The **palCreateTimer ()** routine is called first when requesting the use of a timer. It specifies the function to be called when a timer expires.

Parameters	Type	Description
[in] pFuncCb	palTimerCallback_t	Pointer to a function to call when timer expires
[in] pvUser	void *	Context data passed to timer callback as a parameter
[out] pdwTimerId	palDWORD *	Unique ID to use when using timer routines

Return Value	Type	Description
	palBOOL	palTRUE if successfully created, palFALSE otherwise.

Example

```
#include "pal.h"

palBOOL  palCreateTimer( palTimerCallback_t pFuncCb,
                         void *pvUser,
                         palDWORD *pdwTimerId )
{
    palCHAR yTimerIdx;
    palBOOL  bReturn = palTRUE;

    /* Find the next available timer */
    for( yTimerIdx = 0; yTimerIdx < MAX_PAL_TIMERS; yTimerIdx++ )
    {
        if( TimerCntrlBlock[yTimerIdx].bUsed == palFALSE )
            break;
    }
```


Copyright © 2001 – 2010 S2 Technologies, Inc.

```
    /* return false if all timers are in use */
    if( yTimerIdx == MAX_PAL_TIMERS )
        bReturn = palFALSE;
    else
    {
        /* Indicate timer is in use */
        TimerCntrlBlock[yTimerIdx].bUsed = palTRUE;
        /* set timer callback */
        TimerCntrlBlock[yTimerIdx].pFuncTimerCb = pFuncCb;
        /* Use the table index as the Timer ID.*/
        *pdwTimerId = yTimerIdx;
        /* create the timer */
        rtosCreateTimer( &TimerCntrlBlock[yTimerIdx].TimerId,
                             TimerCallback, yTimerIdx, 0, TIMER_DISABLED );
    }
    return bReturn;
}
```

4.2.2.2. palDeleteTimer()

Delete timer

Prototype

```
palBOOL palDeleteTimer( palDWORD  dwTimerId );
```

Description

The **palDeleteTimer()** routine is called when the user is done with the timer and resources can be released.

Parameters	Type	Description
[in] dwTimerId	palDWORD	Unique Timer ID returned by **palCreateTimer()**

Return Value	Type	Description
	palBOOL	palTRUE if successfully deleted, palFALSE otherwise.

Example

```
#include “pal.h”


palBOOL  palDeleteTimer( palDWORD dwTimerId )
{
    rtosDeleteTimer( timerCntrlBlock[dwTimerId].TimerId );
    timerCntrlBlock[dwTimerId].bUsed = palFALSE;
}
```


Copyright © 2001 – 2010 S2 Technologies, Inc.

4.2.2.3. palStartTimer()

Start timer

Prototype

```
palBOOL palStartTimer( palDWORD  dwTimerId,
                       palDWORD  dwMSec );
```

Description

The **palStartTimer()** routine is called to start a periodic timer based on a previously created unique timer ID.

Parameters	Type	Description
[in] dwTimerId	palDWORD	Unique Timer ID
[in] dwMSec	palDWORD	Periodic timer duration in milliseconds

Return Value	Type	Description
	palBOOL	palTRUE if successfully started, palFALSE otherwise.

Example

```
#include “pal.h”

palBOOL  palStartTimer( palDWORD dwTimerId, palDWORD dwMSec )
{
    palLONG  dwResult;
    palBOOL  bReturn = palTRUE;

    /* make sure timer is in range */
    if( dwTimerId < MAX_PAL_TIMERS )
    {
        dwResult = rtosStartTimer( &TimerCntrlBlock[dwTimerId],
                                   MSEC_TO_TICKS(dwMSec),
                                   TIMER_PERIODIC );

        if( dwResult != RESULT_OK )
            bReturn = palFALSE;
    }
    else
    {
```

Copyright © 2001 – 2008 S2 Technologies, Inc.

```
        /* timer out of range, return false */
        bReturn = palFALSE;
    }
    return bReturn;
}
```

 Copyright © 2001 – 2010 S2 Technologies, Inc.

4.2.2.4. palStopTimer()

Stop timer

Prototype

```
palBOOL palStopTimer( palDWORD  dwTimerId );
```

Description

The **palStopTimer()** routine is called to stop a periodic timer.

Parameters	Type	Description
[in] dwTimerId	palDWORD	Unique Timer ID

Return Value	Type	Description
	palBOOL	palTRUE if successfully stopped, palFALSE otherwise.

Example

```
#include "pal.h"

palBOOL  palStopTimer( palDWORD dwTimerId )
{
    palDWORD dwResult;
    palBOOL  bReturn = palTRUE;

    /* make sure timer is in range */
    if( dwTimerId < MAX_PAL_TIMERS )
    {
        dwResult = rtosStopTimer( &TimerCntrlBlock[dwTimerId] );

        if( dwResult != RESULT_OK )
             bReturn = palFALSE;
    }
    else
    {
        /* timer out of range return false */
        bReturn = palFALSE;
    }
    return bReturn;
}
```

Copyright © 2001 – 2008 S2 Technologies, Inc.

4.2.2.5. palGetTime()

Return system time

Prototype:

```
palDWORD palGetTime( void );
```

Description

The **palGetTime()** routine is implemented to return a system time such as an OS timer tick count. The information in srcfg.h defines the time units (seconds/tick, milliseconds/tick, or microseconds/tick) and duration of each tick (1 unit/tick, 5 units/tick, and etc.).

Parameters	Type	Description
None		

Return Value	Type	Description
	palDWORD	32-bit value of the system timer

Example

```
#include "pal.h"


palDWORD palGetTime( void )
{
    return (palDWORD)rtosGetTime();
}
```


Copyright © 2001 – 2010 S2 Technologies, Inc.

4.2.3. Protection using Mutex

The STRIDE Runtime protects critical data from multiple and simultaneous accesses by using routines that use mutex objects. In case of multi-process target is enabled, shared data should be protected from multiple and simultaneous accesses by multiple processors.

Mutex routines include:

palMutexInit()

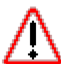
palMutex Destroy()

palMutexLock()

palMutexUnlock()

The PAL requires implementing protection using Mutexes that can be created and, in case of multi-process target, used with a unique name by any STRIDE Runtime module or PAL itself. The STRIDE Runtime guarantees that calls to **palMutexLock()** will not be nested.

These routines must be implemented using mutexes.



These routines are called extensively throughout the Runtime. Extreme care should be taken that an excessive amount of time is not expended during execution of the routine's implementation.

Copyright © 2001 – 2008 S2 Technologies, Inc.

4.2.3.1. palMutexInit()

Initialize a mutex object

Prototype

```
palBOOL palMutexInit( void* *ppvMutex, const palCHAR *szName );
```

Description

The **palMutexInit()** routine initializes a mutex object. In case of multi-process target, mutex should be a named mutex with the unique name of *szName*.

Parameters	**Type**	**Description**
[out] ppvMutex	void* *	Pointer to pointer to a mutex object
[in] szName	const palCHAR *	Unique name for memory segment (can be ignored if multi-process target is not enabled)

Return Value	**Type**	**Description**
	palBOOL	palTRUE on success, palFALSE otherwise

Example

```
#include "pal.h"
#include "srcfg.h"

palBOOL palMutexInit( void* *ppvMutex, const palCHAR *szName )
{
#if srCFG_MULTI_PROC_TARGET
    *ppvMutex = rtosMutexInit(szName);
#else
    *ppvMutex = rtosMutexInit();
#endif

    return palTRUE;
}
```


Copyright © 2001 – 2010 S2 Technologies, Inc.

4.2.3.2. palMutexDestroy()

Destroy a mutex object

Prototype

```
void palMutexDestroy( void* pvMutex );
```

Description

The **palMutexDestroy()** routine destroys a mutex object.

Parameters	Type	Description
[in] pvMutex	void*	Pointer to a mutex object

Example

```
#include "pal.h"

void palMutexDestroy( void* pvMutex )
{
    rtosMutexDestroy(pvMutex);
}
```

4.2.3.3. palMutexLock()

Lock a mutex object

Prototype

```
palBOOL palMutexLock( void* pvMutex );
```

Description

The **palMutexLock()** routine locks a mutex object.

Parameters	Type	Description
[in] pvMutex	void*	Pointer to a mutex object

Return Value	Type	Description
	palBOOL	palTRUE on success, palFALSE otherwise

Example

```
#include "pal.h"


palBOOL palMutexLock( void* pvMutex )
{
    rtosMutexLock(pvMutex);
    return palTRUE;
}
```

 Copyright © 2001 – 2010 S2 Technologies, Inc.

4.2.3.4. palMutexUnlock()

Unlock a mutex object

Prototype

```
palBOOL palMutexUnlock( void* pvMutex );
```

Description

The **palMutexUnlock()** routine unlocks a mutex object.

Parameters	Type	Description
[in] pvMutex	void*	Pointer to a mutex object

Return Value	Type	Description
	palBOOL	palTRUE on success, palFALSE otherwise

Example

```
#include "pal.h"

palBOOL palMutexUnlock( void* pvMutex )
{
    rtosMutexUnlock(pvMutex);
    return palTRUE;
}
```

Copyright © 2001 – 2010 S2 Technologies, Inc.

4.2.4. Memory Management

When the STRIDE Runtime requires memory to be dynamically allocated, it will call the PAL memory routines:

palMemAlloc()

palMemFree()

When the STRIDE Runtime requires internal static memory or, in case of multi-process target, shared memory to be created, opened or closed, it will call the PAL memory segment routines:

palMemSegmentOpen()

palMemSegmentClose()

The **palMemAlloc()** routine must return a pointer to a memory location at least as large as the size requested, and if no memory is available a null pointer should be returned. The size and number of the memory blocks needed is entirely dependent on the following three things:

1. The size of the STRIDE messages defined in your system
2. The number of messages outstanding
3. The memory size defined for your trace logs.

For more information about memory requirements, refer to the *Messaging Memory* section of the *STRIDE Runtime Developer's Guide*.

Implementation of these routines is dependent on how your system uses memory. If dynamic memory is available, the C *malloc()* and *free()* routines or the C++ *new* and *delete* operators can be wrapped by **palMemAlloc()** and **palMemFree().** The RTOS may also use partition-type memory if it is available.

In case of multi-process target, dynamic memory will be shared among multiple applications. Therefore, these routines should simply call STRIDE Runtime Memory Management module *srMem*. The *srMem* will be dependent on shared memory that will be allocated by calling PAL memory segment routines **palMemSegmentOpen()** and **palMemSegmentClose()** described below.

The STRIDE Runtime may use **palMemAlloc()** and **palMemFree()** frequently, so it is important that an efficient memory management scheme is used.

The **palMemSegmentOpen()** routine must return a pointer to a memory segment at least as large as the size requested, and if no memory is available a null pointer should be returned. The size and number of the memory blocks needed is entirely dependent on whether the STRIDE Runtime memory management and multi-process target are enabled or not.

 Copyright © 2001 – 2010 S2 Technologies, Inc.

In case of multi-process target, dynamic, configurable, and internal static memory will be allocated through **palMemSegmentOpen()**. The size of static memory, used by Runtime for internal bookkeeping purposes, depends on the STRIDE Runtime module data. However, the sizes and maximum number of memory blocks can be configured in *srcfg.h*.

The STRIDE Runtime Memory Management module *srMem* determines the sizes and maximum number of memory blocks based on the parameters in *srcfg.h*. Refer to the **STRIDE Runtime Developer's Guide** (available through STRIDE Online Help) for more details about *srcfg.h*.

In case of multi-process target is enabled, implementation of **palMemSegmentOpen()** and **palMemSegmentClose()** routines is dependent on how your system uses shared memory. The typical and mostly available method is to use Memory-Mapped Files (MMF).

Copyright © 2001 – 2010 S2 Technologies, Inc.

4.2.4.1. palMemAlloc()

Allocate dynamic memory

Prototype

```
void* palMemAlloc( palWORD wSize );
```

Description

The **palMemAlloc()** routine is used to allocate dynamic memory. It returns a pointer to a buffer at least *wSize* bytes in length. If the allocation fails, a value of NULL should be returned. This routine can be called frequently, so it is important that an efficient memory management scheme is used. In case of multi-process target, this routine should simply call STRIDE Runtime routine *_srMem_Allocate()*.

Parameters	Type	Description
[in] wSize	palWORD	Size in bytes of the memory request

Return Value	Type	Description
	void*	Pointer to the block of memory allocated on success, or 0 on failure

Example

```
#include “pal.h”
#include "srcfg.h"
#if srCFG_MEMORY_MANAGEMENT
#include "srmem.h"
#endif

void* palMemAlloc( palWord wSize )
{
#if srCFG_MULTI_PROC_TARGET
    return _srMem_Allocate(wSize);
#else
    return rtosMalloc(wSize);
#endif
}
```


Copyright © 2001 – 2010 S2 Technologies, Inc.

4.2.4.2. palMemFree()

Free allocated memory

Prototype

```
palBOOL palMemFree( void* pvMem );
```

Description

The **palMemFree()** routine is used to release memory allocated with **palMemAlloc().** In case of multi-process targetthis routine should simply call STRIDE Runtime routine *_srMem_Free()*.

Parameters	Type	Description
[in] pvMem	void*	Address of memory to release

Return Value	Type	Description
	palBOOL	palTRUE on success, palFALSE otherwise

Example

```
#include “pal.h”
#include "srcfg.h"
#if srCFG_MEMORY_MANAGEMENT
#include "srmem.h"
#endif

palBOOL  palMemFree( void *pvMem )
{
#if srCFG_MULTI_PROC_TARGET
    _srMem_Free(pvMem);
#else
    rtosFree(pvMem);
#endif
    return palTRUE;
}
```

4.2.4.3. palMemSegmentOpen()

Create or open memory segment

Prototype

```
void* palMemSegmentOpen( const palCHAR* szName,
                         palDWORD dwSize,
                         palBOOL* pbFirstInst );
```

Description

The **palMemSegmentOpen ()** routine is used to create or open a memory segment. It returns a pointer to a buffer at least d*wSize* bytes in length. If the creation or opening fails, a value of NULL should be returned. This routine should create a memory segment with a unique name of *szName* when called for the first time. When called subsequently with the same name, this routine should open the existing memory segment. In case of multi-process target, the STRIDE Runtime calls this routine for creation and opening of dynamic, configurable, and internal static memory. Furthermore, the implementation of these routines is dependent on how your system uses shared memory. The typical and mostly available method to implement is shared memory is to use Memory-Mapped Files (MMF). This routine can optionally count the number of opened instances for the purpose of unlinking in **palMemSegmentClose()**.

Parameters	Type	Description
[in] szName	const palCHAR*	Unique name for memory segment to create or open
[in] dwSize	palDWORD	Size in bytes of memory segment request
[in] pbFirstInst	palBOOL*	palTRUE if memory segment was created for the first time. palFALSE if not (previously created segment was opened).

Return Value	Type	Description
	void*	Starting address of memory segment created or opened on success, or 0 on failure

Example

```
#include “pal.h”
#include "srcfg.h"


/* Table of handles for shared memory */
```


Copyright © 2001 – 2010 S2 Technologies, Inc.

```
typedef struct
{
    palCHAR  szName[PAL_MODULE_STR_SIZE];
    void*    pvMem;
    palLONG  lFileDes;
    palDWORD dwSize;
} _palMemHandlesInfo_t;

static _palMemHandlesInfo_t tMemHandlesInfo[32] = {{{0}, palNULL, -1, 0}};

void* palMemSegmentOpen( const palCHAR* szName,
                         palDWORD dwSize,
                         palBOOL* pbFirstInst )
{
    void *pvMem;
    palWORD wIndex;
#if srCFG_MULTI_PROC_TARGET
    palLONG lFileDes;
#endif

    /* determine a free slot in handles table for later release */
    for (wIndex = 0;
         wIndex < sizeof(tMemHandlesInfo)/sizeof(tMemHandlesInfo[0]);
         wIndex++)
    {
        if (tMemHandlesInfo[wIndex].szName[0] == 0)
            break;
    }
    if (wIndex == sizeof(tMemHandlesInfo)/sizeof(tMemHandlesInfo[0]))
    {
        palLog(palLOG_LEVEL_ERROR, " palMemSegmentOpen: no free slots");
        return palNULL;
    }

#if srCFG_MULTI_PROC_TARGET
    /* try to create a new shared memory object */
    lFileDes = rtosShm_open(szName);

    if (lFileDes == -1)
    {
```

```
        *pbFirstInst = palFALSE;
        /* open existing shared memory object */
        lFileDes = rtosShm_open(szName);

        if (lFileDes == -1)
        {
            palLog(palLOG_LEVEL_ERROR, "palMemSegmentOpen: shm open failed.");
            return palNULL;
        }
    }
    else
    {
        if (rtosFtruncate(lFileDes, dwSize+1) != 0)
        {
            rtosCloseHandle(lFileDes);
            if (*pbFirstInst == palTRUE)
                rtosShm_unlink(szName);
            palLog(palLOG_LEVEL_ERROR, "palMemSegmentOpen: truncate failed.");
            return palNULL;
        }
        *pbFirstInst = palTRUE;
    }

    /* map shared memory object */
    pvMem = rtosMmap(0, dwSize+1, lFileDes, 0);

    if (!pvMem)
    {
        rtosCloseHandle(lFileDes);
        if (*pbFirstInst == palTRUE)
            rtosShm_unlink(szName);
        palLog(palLOG_LEVEL_ERROR, "palMemSegmentOpen: mmap failed.");
        return palNULL;
    }

    /* count the number of opened instances for later unlinking */
    (*(char*)pvMem)++;
    pvMem = (char*)pvMem + 1;

    /* add handles to table for later release */
```


Copyright © 2001 – 2010 S2 Technologies, Inc.

```
    strncpy(tMemHandlesInfo[wIndex].szName, szName, 128);
    tMemHandlesInfo[wIndex].pvMem = pvMem;
    tMemHandlesInfo[wIndex].lFileDes = lFileDes;
    tMemHandlesInfo[wIndex].dwSize = dwSize;

#else /* NOT srCFG_MULTI_PROC_TARGET */
    pvMem = rtosMalloc(dwSize);
    if (pvMem == NULL)
    {
        palLog(palLOG_LEVEL_ERROR, " palMemSegmentOpen: malloc failed.");
        return palNULL;
    }
    *pbFirstInst = palTRUE;
#endif /* srCFG_MULTI_PROC_TARGET */

    return pvMem;
}
```

4.2.4.4. palMemSegmentClose()

Close created memory segment

Prototype

```
palBOOL palMemSegmentClose( void *pvMem );
```

Description

The **palMemSegmentClose()** routine is used to close memory segments created with **palMemSegmentOpen().** This routine can optionally count the number of closed instances and unlink the memory segment accordingly.

Parameters	Type	Description
[in] pvMem	void*	Starting address of memory segment to close

Return Value	Type	Description
	palBOOL	palTRUE on success, palFALSE otherwise

Example

```
#include “pal.h”
#include "srcfg.h"

/* See palMemSegmentOpen() example */

palBOOL palMemSegmentClose( void *pvMem )
{
#if srCFG_MULTI_PROC_TARGET
    palWORD wIndex;
    for (wIndex = 0;
         wIndex < sizeof(tMemHandlesInfo)/sizeof(tMemHandlesInfo[0]);
         wIndex++)
    {
        if (tMemHandlesInfo[wIndex].pvMem == pvMem)
        {
            /* count number of closed instances; if zero instances, unlink */
            tMemHandlesInfo[wIndex].pvMem =
(char*)tMemHandlesInfo[wIndex].pvMem - 1;
            (*(char*)tMemHandlesInfo[wIndex].pvMem)--;
            if (*(char*)tMemHandlesInfo[wIndex].pvMem == 0)
```

 Copyright © 2001 – 2010 S2 Technologies, Inc.

```
                rtosShm_unlink(tMemHandlesInfo[wIndex].szName);

            rtosMunmap(tMemHandlesInfo[wIndex].pvMem,
                       tMemHandlesInfo[wIndex].dwSize);
            tMemHandlesInfo[wIndex].pvMem = NULL;
            rtosCloseHandle(tMemHandlesInfo[wIndex].lFileDes);
            tMemHandlesInfo[wIndex].lFileDes = -1;
            tMemHandlesInfo[wIndex].szName[0] = 0;
            break;
        }
    }
#else /* NOT srCFG_MULTI_PROC_TARGET */
    if (pvMem != NULL)
        rtosFree(pvMem);
#endif /* srCFG_MULTI_PROC_TARGET */
    return palTRUE;
}
```

4.2.5. Logging

The STRIDE Runtime as well as PAL routines may use PAL logging routine to report errors, warnings, trace information and debug information.

palLog()

Implementing the **palLog()** routine is optional.

 Copyright © 2001 – 2010 S2 Technologies, Inc.

4.2.5.1. palLog()

Log messages

Prototype

```
void palLog( palLogLevel_e eLevel, const palCHAR* szFmt, ... );
```

Description

The **palLog()** routine is used to log messages from STRIDE Runtime and PAL itself to report errors, warnings, trace information and debug information according to log-level. This routine can optionally implement variable argument list to format the message string. Implementing of this routine is optional.

Parameters	Type	Description
[in] eLevel	palLogLevel_e	Log level
[in] szFmt	const palCHAR*	Formatted string for message
[in] …	(Optional)	Variable argument list to format *szFmt*

```
typedef enum
{
    palLOG_LEVEL_TRACE,
    palLOG_LEVEL_DEBUG,
    palLOG_LEVEL_INFO,
    palLOG_LEVEL_WARN,
    palLOG_LEVEL_ERROR,
    palLOG_LEVEL_FATAL
} palLogLevel_e;
```

Example

```
#include “pal.h”
/* to handle variable argument list (optional) */
#include <stdio.h>
#include <stdarg.h>

void palLog( palLogLevel_e eLevel, const palCHAR* szFmt, ... )
{
    palCHAR szMsg[4096] = {0};
    va_list args;

    if (szFmt == palNULL)
        return;
```

```
    va_start(args, szFmt);
    vsnprintf(szMsg, sizeof(szMsg)/sizeof(szMsg[0])-1, szFmt, args);
    va_end(args);

    if (eLevel >= palLOG_LEVEL_WARN)
    {
        fputs(eLevel>=palLOG_LEVEL_ERROR ? "ERROR: " : "WARNING: ", stderr);
        fputs(szMsg, stderr);
        fputs("\n", stderr);
        fflush(stderr);
    }
    else
    {
        fputs(szMsg, stdout);
        fputs("\n", stdout);
        fflush(stdout);
    }
}
```

 Copyright © 2001 – 2010 S2 Technologies, Inc.

4.3. PAL IO Services

The PAL IO Services consists of a set of functions required by the STRIDE Runtime to send data to and receive data from your target platform. The Runtime has been designed to be independent of any specific transport mechanism.

Full implementation of the following PAL IO functions is required:

Function Name	Description
Transmit	**page 62**
palOutPndReg	Register a routine to query for pending I-blocks
palOutRdyReg	Register a routine to signal transport ready to transmit
palOut	Transmit a buffer
Receive	**page 68**
palInReg	Register a routine to be called when data is available

4.3.1. Function Registration

Many of the IO services require the PAL to call functions in the STRIDE Runtime when the STRIDE Runtime calls functions in the PAL. In order to allow the PAL to call these functions without creating dependencies on the Runtime, the PAL must implement several registration routines that allow it to store references to these Runtime routines. The Runtime calls these registration routines only once, during its initialization. The references, or callbacks, can then be called when appropriate to incorporate your transport into the PAL.

The Runtime calls **palOutPndReg()** to allow the PAL to store a function pointer to a routine that can be called to query the number of messages the Runtime has ready to transmit. Calling the registered callback routine is optional; it enables you to check if any messages are ready to be transmitted.

The Runtime calls **palOutRdyReg()** to allow the PAL to store a function pointer to a routine that can be called to signal to the Runtime that your transport is ready for the next message to be transmitted. The Runtime will not transmit a message until this registered callback is called. Because of this, the callback can be used as a mechanism to control when data is transmitted over your transport (useful for half-duplex transports).

The Runtime calls the **palInReg()** routine to allow the PAL to store a pointer to a routine called to input a message received to the Runtime. When your transport mechanism detects a complete STRIDE message, a call to the registered callback should be made.

Copyright © 2001 – 2010 S2 Technologies, Inc.

4.3.2. Transmit Data

The required calling sequence between the Runtime and the PAL to transmit data is shown in Figure 6 and detailed below.

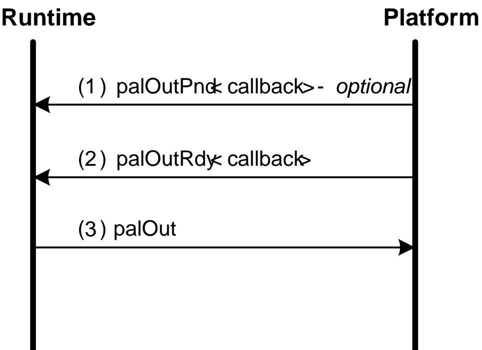


Figure 6. Transmit Data Sequence

Transmit Data Sequence

1. The PAL can call the registered callback routine for **palOutPndReg()** to query the current output queue of the Runtime. The pending routine is helpful for half-duplex systems in determining when and how long to open up the transmit channel.
2. The PAL is required to call the registered callback routine for **palOutRdyReg()**. This routine is called when the transport is ready to transmit data. The Runtime requires the routine to be called before each output transaction, allowing the PAL to optionally control when data is sent out. To avoid having the Runtime queue data while waiting for the ready routine to be called, insert a call to the ready routine into **the** palOut() routine, so that the ready routine is called once for every call **to** palOut().
3. The **palOut()** routine is called to instruct the PAL to transmit a packet of data using its transport services.

The Transmit routines include:

palOutPndReg()

palOutRdyReg()

palOut()

Copyright © 2001 – 2010 S2 Technologies, Inc.

4.3.2.1. palOutPndReg()

Query Runtime output queue

Prototype

```
typedef palWORD ( *palOutPndCallback_t )( void );
palOutPndCallback_t palOutPndReg( palOutPndCallback_t pFuncCb );
```

Description

The STRIDE Runtime calls the **palOutPndReg()** routine once during Runtime initialization to register the callback with the PAL. The callback can then be used to query the current output queue of the Runtime for data ready to be transmitted.

Parameters	Type	Description
[in] pFuncCb	palOutPndCallback_t	Function callback to query Out data pending

Return Value	Type	Description
	palOutPndCallback_t	The previously registreted callback or palNULL if not such set.

Example

```
#include "pal.h"

palOutPndCallback_t ptOutPendingCb = palNULL;

palOutPndCallback_t  palOutPndReg( palOutPndCallback_t pFuncCb )
{
    palOutPndCallback_t ptOldFuncCb = ptOutPendingCb;
    ptOutPendingCb = pFuncCb;
    return ptOldFuncCb;
}

/*
 * if transport is half-duplex we may need to check if we can release it yet
 */
 palBool halfDuplexIsReady( int duplexId )
{
    palWord nblks = 0;
    nblks = ptOutPendingCb();
    if( nblks > 0 )
```

```
    {
        return palFALSE;
    }
    return palTRUE;
}
```

The **palOutPndReg()** routine is used to dynamically bind a service offered by the Runtime for the PAL. The callback routine functionality is specified below.

Prototype

```
palWORD ( *palOutPndCallback )( void );
```

Parameters	Type	Description
None		

Return Value	Type	Description
	palWORD	The number of separate data blocks to send or 0 when no data is pending to transmit

 Copyright © 2001 – 2010 S2 Technologies, Inc.

4.3.2.2. palOutRdyReg()

Indicate transport ready to transmit

Prototype

```
typedef void ( *palOutRdyCallback_t ) ( void );
palOutRdyCallback_t palOutRdyReg( palOutRdyCallback_t pFuncCb );
```

Description

The STRIDE Runtime calls the **palOutRdyReg()** routine once during Runtime initialization to register the callback with the PAL. The callback is called when the transport is ready to transmit data. The Runtime requires the callback routine to be called before each output transaction, in order to allow the PAL to optionally control when data is sent out.

Parameters	Type	Description
[in] pFuncCb	palOutRdyCallback_t	Indicates that transport is ready to receive the next packet of data

Return Value	Type	Description
	palOutRdyCallback_t	The previously registreted callback or palNULL if not such set.

Example

```
#include "pal.h"

palOutRdyCallback_t ptOutRdyCb = palNULL;

palOutRdyCallback_t palOutRdyReg( palOutRdyCallback_t pFuncCb )
{
    palOutPndCallback_t ptOldFuncCb = ptOutRdyCb;
    ptOutRdyCb = pFuncCb;
    return ptOldFuncCb;
}

palBOOL  palOut( const palBYTE *pyOutBuffer, palWORD wSize )
{
    /* signal ready to transmit next buffer, no need to throttle */
    ptOutRdyCb();
    /* send I-Block */
    rtosTransportWrite( pyOutBuffer,wSize );
    return palTRUE;
```

```
}
```

The **palOutRdyReg()** routine is used to dynamically bind a service offered by the Runtime for the PAL. The callback routine functionality is specified below.

Prototype

```
void ( *palOutRdyCallback ) ( void );
```

Parameters	Type	Description
None		

Return Value	Type	Description
None		

Copyright © 2001 – 2010 S2 Technologies, Inc.

4.3.2.3. palOut()

Send data to host

Prototype

```
palBOOL palOut( const palBYTE *pyOutBuffer, palWORD wSize )
```

Description

The **palOut()** routine is called by the STRIDE Runtime to send data to over the target transport mechanism to the host platform. A buffer containing the data address and size are passed as parameters. For more about Simplified Link Application Protocol (SLAP), see *S2 SLAP Package* on page 73.

Parameters	Type	Description
[in] pyOutBuffer	const palBYTE*	Address of the data to transmit
[in] wSize	palWORD	Size in bytes of the data

Return Value	Type	Description
	palBOOL	palTRUE on successfully received data to send, palFALSE otherwise

Example

```
#include "pal.h"
#include "s2slap.h"
palBOOL  palOut( const palBYTE *pyOutBuffer, palWORD wSize )
{
    /* runtime already signaled for ready to send next data */
    /* send I-Block */
    rtosTransportWrite( pyOutBuffer,wSize );
    return palTRUE;
}
```

4.3.3. Receive Data

The required calling sequence to receive data from the PAL between the Runtime and the PAL is defined in Figure 7 and below.

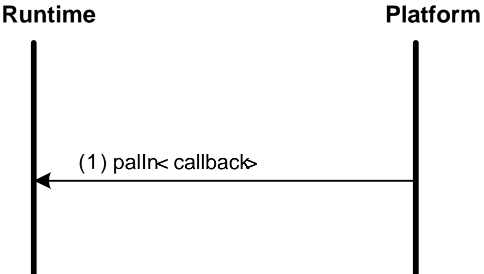


Figure 7. Receive an I-Block Sequence

Receive Data Sequence

1. The PAL is calls the registered callback routine from **palInReg().** This routine is called when data has been extracted from the transport and identified for the Runtime.

The Receive routine:

palInReg()

 Copyright © 2001 – 2010 S2 Technologies, Inc.

4.3.3.1. palInReg()

Indicates data ready for Runtime

Prototype

```
typedef void (*palInDataCallback_t)( const palBYTE *pyInBuffer,
                                     palWORD  wSize );
palInDataCallback_t palInReg( palInDataCallback_t pFuncCB );
```

Description

The **palInReg()** routine is called when data has been extracted from the transport and identified for the Runtime.

Parameters	Type	Description
[in] pFuncCB	palInDataCallback_t	Routine to call when data is received from transport

Return Value	Type	Description
	palInDataCallback_t	The previously registreted callback or palNULL if not such set.

Example

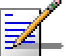
```
#include "pal.h"

palInDataCallback_t ptInDataCb = palNULL;

palInDataCallback_t palInReg( palInDataCallback_t pFuncCb )
{
    palOutPndCallback_t ptOldFuncCb = ptInDataCb;
    ptInDataCb = pFuncCb;
    return ptOldFuncCb;
}

/* transport has data ready to hand to the Runtime */
void transportDataReady( const palBYTE *pyDataBuffer, palWORD wSize )
{
    ptInDataCb( pyDataBuffer, wSize );
}
```

Copyright © 2001 – 2010 S2 Technologies, Inc.



The **palInReg()** routine is used to dynamically bind a service offered by the Runtime for the PAL. The callback routine functionality is specified below.

Prototype

```
void ( *palInDataCallback ) ( const palBYTE *pyBuffer, palWORD wSize );
```

Parameters	**Type**	**Description**
[in] pyBuffer	const palBYTE *	Buffer containing data received from transport
[in] wSize	palWORD	Size of buffer

Return Value	**Type**	**Description**
None		

Copyright © 2001 – 2010 S2 Technologies, Inc.

5. Advanced PAL Usage

5.1. Using RFC Proxy Routines

Remote function calls (RFCs) that are located on the host but called from the target are supported by a call from an RFC Proxy routine rather than the actual function. The RFC Proxy routine creates a temporary NID, and then calls **palWait()** while waiting for the return of the remote function call. The STRIDE Runtime calls **palNotify()** with the same NID when the function call return value is ready. Unlike with the Runtime thread and RFC Stub threads, this wait occurs within the context of the application thread making the RFC Proxy call, and the notification is delivered to that thread.

Threads that call RFC Proxy routines may also have the need to wait and be notified for other reasons. For example, two threads in an application may synchronize access to a piece of shared memory, or application threads may be sending and receiving messages using the STRIDE Messaging Model. In both cases, when the threads also make RFC Proxy calls, the threads will have multiple reasons to be waiting on synchronization primitives, or NIDs.

Problems can arise when NIDs are not unique to each instance a thread may be required to wait and be notified. As an example, if a NID was simply the ID of a thread, and the thread would wait and be notified based on that ID, then any notification of that NID would wake up the waiting thread. If the thread was waiting on an RFC Proxy call, and its ID was notified by another task synchronizing with it for access to shared memory, then the thread would wake up and continue as if the RFC Proxy call completed, even though it did not. To avoid this problem, separate NIDs are required for synchronizing access to shared memory, and for waiting on the RFC Proxy call.

In general, a unique NID must be available for each reason a thread may need to wait and be notified. It is for this reason that Proxy NIDs are created and deleted with a different interface, **palCreateRFCProxyNID()** and **palDeleteRFCProxyNID(),** from other NIDs. Depending on the types of synchronization primitives available in your system, RFC Proxy NIDs may need to be different than NIDs created with **palCreateNID()** and **palDeleteNID().**

The different NIDs may be references to different types of notification primitives, or they may simply provide enough information so that **palNotify()** can generate proper notifications.

In either case, as RFC Proxy routines are called, attention must be paid to the NIDs used to notify threads.

5.2. Using the PAL without the RTOS/Scheduler

It is possible to use the PAL without an RTOS, as long as the services required by the PAL can be provided within the target environment. The STRIDE Runtime and RFC Stub Threads must be properly initialized, and then scheduled with the rest of the system.

The target environment must ensure the STRIDE Runtime thread, RFC Stub threads and any application threads using the STRIDE Messaging Model or RFC Proxy calls are

scheduled. The Runtime and RFC Stub threads must also be initialized prior to scheduling any STRIDE thread. See the STRIDE Runtime Developer's Guide for more information about the Runtime initialization and API.

The **palWait()** routine may not need to be implemented if there is no chance for scheduler preemption.

A mechanism to notify threads of pending messages must be provided in the **palNotify()** routine.

If there is no chance for preemption of a thread, then critical data protection services, **mutex** routines**,** do not need to be provided. Calling STRIDE Runtime or PAL functions from Interrupt Service Routines (ISRs) may cause preemption of other STRIDE calls. For more information, refer to *Calling from an ISR* on page 72. If calling STRIDE Runtime and PAL functions from ISRs is required, **mutex** routines need to be fully implemented.

5.3. Calling from an ISR

Calling STRIDE Runtime routines from an Interrupt Service Routine (ISR) is generally not recommended, unless all of the services of the PAL can be accessed from the ISR context. Any Runtime routine may ultimately call on PAL services.

Timers, critical data protection and notification are the most common areas of the PAL that may conflict with being called from the ISR context. Workarounds include using software-synchronous timers and notification, as well as disabling interrupts for critical data protection

Copyright © 2001 – 2010 S2 Technologies, Inc.

6. S2 SLAP Package

6.1. Overview

The Simplified Link Application Protocol (SLAP) is a link protocol that is used to transmit and receive frames of data between two platforms. The sole purpose of the SLAP is to guarantee that frames are successfully transmitted between the two platforms.

The SLAP verifies the integrity of the data contained within the frame and is able to resynchronize quickly in the event of missed frames. This is accomplished through the use of "data stuffing".

6.1.1. SLAP Frame

The figure below shows the overall structure of the SLAP frame. The header contains 6 octets used for framing logic. The header uses 8- and 16-bit data elements in defining the frame and follows little-endian byte ordering. In a little-endian architecture, the bytes are transmitted least significant byte (LSB) first for a 16-bit value. The data section can contain between zero (0) and 65,527 octets. The maximum size of a SLAP frame is 65,535 octets.

Header	Frame Data
6 octets	**0 – 65527 octets**

6.1.2. SLAP Frame Header Format

In the SLAP frame format, shown below, each frame starts with a single octet preamble consisting 0x7E. The preamble signifies the beginning of a SLAP frame.

SLAP uses octet data stuffing in order to allow fast synchronization to the start of frames. A single octet with the value of the PREAMBLE_CHAR appears only at the start of a SLAP frame. Any other octet equal to the PREAMBLE_CHAR will be preceded by the ESC_CHAR with the original octet XOR'd with the XOR_CHAR. This ensures that the Preamble character will only be seen at the start of a SLAP frame. Any octet in the SLAP frame that has a value of the ESC_CHAR is preceded by the ESC_CHAR with the original octet XOR'd with the XOR_CHAR.

Copyright © 2001 – 2010 S2 Technologies, Inc.

6.1.3. Data Stuffing Characters

7 6 5 4 3 2 1 0
Preamble (0x7E)
Frame Type
Length (low byte)
Length (high byte)
Checksum(low byte)
Checksum(high byte)
Data
Data
...

Copyright © 2001 – 2010 S2 Technologies, Inc.

Char	Value	Comment
PREAMBLE_CHAR	0x7e	Large non-negative 8-bit value
ESC_CHAR	0x7d	Large non-negative 8-bit value
XOR_CHAR	0x20	Turns 0x7e to 0x5e and 0x7d to 0x5d

The values of the PREAMBLE_CHAR and the ESC_CHAR are selected in order to minimize the likelihood of data octets having the same values.

6.1.4. Examples of Data Stuffing

0x7e is transmitted as 0x7d, 0x5e (0x7e XOR'd with 0x20 = 0x5e)

0x7d is transmitted as 0x7d, 0x5d (0x7d XOR'd with 0x20 = 0x5d)

Copyright © 2001 – 2010 S2 Technologies, Inc.

6.2. SLAP Services

See s2slap.h on page 156 and s2slap.c on page 87.

The Slap Services routines include:

- **s2slapTxMsgFormat()**
- **s2slapRXMsgExtract()**
- **s2slapRegDataReadyCb()**
- **s2slapRegErrorCb()**

 Copyright © 2001 – 2010 S2 Technologies, Inc.

6.3. *s2slapTxMsgFormat*

Format SLAP frame

Prototype

```
palBOOL s2slapTxMsgFormat (palBYTE  *pyDest,
                           palWORD   wDestSize,
                           palWORD  *pwDestSize,
                           const palBYTE  *pySrc,
                           palWORD   wSrcSize
                           palBYTE   yType);
```

Description

The **s2slapTxMsgFormat()** routine is used to format a buffer into a SLAP frame prior to transmitting the data across a link. It places header information, calculates a checksum, and adds any data stuffing characters. If the destination buffer is too small to hold the entire frame, the function returns a false value and the *pwDestSize* parameter is set to the size needed to hold the completed SLAP frame.

Parameters	Type	Description
[out] pyDest	palBYTE *	Destination for the SLAP frame
[in] wDestSize	palWORD	Size of destination buffer
[out] pwDestSize	palWORD *	Size of SLAP frame
[in] pySrc	const palBYTE *	Data to frame
[in] wSrcSize	palWORD	Size of data
[in] yType	palBYTE	Payload type

Return Value	Type	Description
	palBOOL	palTRUE if pyDest contains a valid SLAP frame, palFALSE when pyDest is too small to hold the SLAP frame. pwDestSize indicates the size needed

Example

```
void TransmitFrame(palBYTE *pyBuffer, palWORD wSize)
{
    palBYTE *pySLAPFrame;
    palWORD wOutSize;

    /* Use 0 as the destination size to see how big the buffer needs to be */
    s2slapTxMsgFormat(NULL, 0, &wOutSize, pyBuffer, wSize,
S2_SLAP_TYPE_IBLOCK);

    /* allocate new buffer for SLAP frame */
    pySLAPFrame = palMemAlloc(wOutSize);

    /* Format SLAP Frame from data buffer */
    s2slapTxMsgFormat(pySLAPFrame, wOutSize, &wOutSize, pyBuffer, wSize,
S2_SLAP_TYPE_IBLOCK);

    /* Send Frame across transport */
    myTransportSend(pySLAPFrame, wOutSize);

    /* Free frame buffer */
    palMemFree(pySLAPFrame);
}
```


Copyright © 2001 – 2010 S2 Technologies, Inc.

6.4. *s2slapRxMsgExtract*

Extract a data buffer from SLAP frame

Prototype

```
palBOOL s2slapRxMsgExtract (palBYTE  *pyDest,
                            palWORD   wDestMaxSize,
                            const palBYTE  *pySrc,
                            palWORD   wSrcSize);
```

Description

The **s2slapRxMsgExtract()** routine is used to extract a data buffer from a SLAP Frame. Since data may become segmented across a transport, this routine handles multiple calls with different-sized pieces of data. When it finds a complete frame, it calls **s2slapDataReady()** with the data buffer. For example, if two frames (sizes 12 bytes and 7 bytes) are transmitted across a link that sends 5 bytes at a time, the first two 5-byte segments will contain an incomplete frame 1. The third segment will contain part of frame 1 and part of frame 2. After the third segment, **s2slapDataReady()** is called with the data from frame 1. After the fourth segment, it is called again with frame 2.

Alternatively, if a single buffer containing multiple frames is passed into **s2slapTxMsgExtract(), s2slapDataReady()** is called after each complete frame is parsed.

Parameters	**Type**	**Description**
[out] pyDest	palBYTE *	Destination buffer
[in] wDestMaxSize	palWORD	Size of destination buffer
[in] pySrc	const palBYTE *	Data from SLAP frame
[in] wSrcSize	palWORD	Size of data

Return Value	**Type**	**Description**
	palBOOL	palTRUE if no errors encountered during buffer extract, palFALSE otherwise.

Example

```
/* Destination Buffer */
#define MAX_DEST_SIZE 1024
palBYTE yDestBuffer[MAX_DEST_SIZE];

/* routine called when data is received by transport */
void ReceiveData(palBYTE *pyBuffer, palWORD wSize)
{
    /* extract data */
    s2slapRxExtract(yDestBuffer, MAX_DEST_SIZE, pyBuffer, wSize);
}
```


Copyright © 2001 – 2010 S2 Technologies, Inc.

6.5. s2slapGetDataReadyCb

Extracted data ready

Prototype

```
typedef void ( *s2slapDataReadyCb_t )( const palBYTE *pyDataBuffer,
                                       palWORD  wSize
                                       palBYTE  yType);
s2slapDataReadyCb_t s2slapRegDataReadyCb(s2slapDataReadyCb_t ptFuncCb);
```

Description

The **s2slapDataReady()** routine is called by **s2slapRxExtract()** when data from a complete SLAP frame is available. This is a user-implemented routine.

Parameters	Type	Description
[in] ptFuncCb	s2slapDataReadyCb_t	Indicates that data is extracted.

Return Value	Type	Description
	s2slapDataReadyCb_t	The previously registreted callback or palNULL if not such set.

Prototype

```
void ( *s2slapDataReadyCallback ) ( const palBYTE * pyDataBuffer,
                                    palWORD wSize,
                                    palBYTE  yType );
```

Parameters	Type	Description
[in] pyDataBuffer	const palBYTE *	Buffer containing extracted data
[in] wSize	palWORD	Size in bytes of buffer
[in] yType	palBYTE	Type of data is contained in buffer

Return Value	Type	Description
None		

Copyright © 2001 – 2010 S2 Technologies, Inc.

Appendix A: pal.h

```
/*****************************************************************************
 *
 * FILE NAME: pal.h
 *
 * DESCRIPTION:
 *   Platform Abstraction Layer (PAL)
 *
 * ---------------------------------------------------------------------------
 * Copyright 2001 - 2008 by S2 Technologies, Inc.
 * ---------------------------------------------------------------------------
 ****************************************************************************/

#ifndef PAL_H
#define PAL_H

#ifdef __cplusplus
extern "C" {
#endif


#define palEXPORT

/*****************************************************************************
 *
 * Primitive Types
 *
 *****************************************************************************/

typedef char            palCHAR;  /* signed 8 bits
*/
typedef unsigned char   palBYTE;  /* unsigned 8 bits
*/
typedef short           palSHORT; /* at least signed 16 bits and no longer than palLONG
*/
typedef unsigned short  palWORD;  /* at least unsigned 16 bits and no longer than palLONG
*/
typedef long            palLONG;  /* at least signed 32 bits
*/
typedef unsigned long   palDWORD; /* at least unsigned 32 bits
*/
typedef unsigned char   palBOOL;  /* able to support palTRUE and palFALSE
*/

#define palFALSE        0
#define palTRUE         1
#define palNULL         0

/*****************************************************************************
 *
 * Operating System
 *
 *****************************************************************************/

/*****************************************************************************
 * Task Synchronization
 ****************************************************************************/

#define palNID_RESERVED_0       0xFFFFFFFF
#define palNID_RESERVED_1       0xFFFFFFFE

#define palSTOP_EVENT           0x80000000
#define palMBOX_EVENT(box)      (palMBOX_EVENTS_MASK & (1 << (box)))

#define palMBOX_EVENTS_MASK     0x000000FF
#define palCUSTOM_EVENTS_MASK   0x0FFFFF00
#define palSYSTEM_EVENTS_MASK   0xF0000000
#define palALL_EVENTS_MASK      (palSYSTEM_EVENTS_MASK | palMBOX_EVENTS_MASK |
palCUSTOM_EVENTS_MASK)
```

 Copyright © 2001 – 2010 S2 Technologies, Inc.

```
/**
 * PAL interface to create a notification Id to syncronize objects.
 * @param pdwNID  [out] Notification Id. The Id returned is used by palWait() and
palNotify().
 * @return  palTRUE on success, palFALSE otherwise.
 */
palEXPORT palBOOL palCreateNID( palDWORD *pdwNID );

/**
 * PAL interface to delete a notification Id used to syncronize objects.
 * @param dwNID  [in] Notification Id.
 * @return  palTRUE on success, palFALSE otherwise.
 */
palEXPORT palBOOL palDeleteNID( palDWORD dwNID );

/**
 * PAL interface to create a RFC proxy notification Id.
 * @param pdwNID  [out] Notification Id.
 * @return  palTRUE on success, palFALSE otherwise.
 */
palEXPORT palBOOL palCreateRFCProxyNID( palDWORD *pdwNID );

/**
 * PAL interface to delete a RFC proxy notification Id.
 * @param dwNID  [in] Notification Id.
 * @return  palTRUE on success, palFALSE otherwise.
 */
palEXPORT palBOOL palDeleteRFCProxyNID( palDWORD dwNID );

/**
 * PAL interface to wait for an event.
 * @param dwNID  [in] Notification Id.
 * @param pdwEvents  [in/out] Events notification where palSTOP_EVENT on stop, Bits 0-7
for mail boxes and the rest custom specified.
 * @return  palTRUE on success, palFALSE otherwise.
 */
palEXPORT palBOOL palWait( palDWORD dwNID, palDWORD *pdwEvents );

/**
 * PAL interface to signal event pending.
 * @param dwNID  [in] Notification Id.
 * @param dwEvents  [in] Events to notify where palSTOP_EVENT for stop, Bits 0-7 for mail
boxes and the rest custom specified.
 * @return  palTRUE on success, palFALSE otherwise.
 */
palEXPORT palBOOL palNotify( palDWORD dwNID, palDWORD dwEvents );

/**
 * PAL interface to return current thread Id.
 * @return  Thread id.
 */
palEXPORT palDWORD palGetThreadId( void );

/**
 * PAL interface to return current process Id.
 * @return  Process id.
 */
palEXPORT palDWORD palGetProcessId( void );

/**
 * PAL interface to suspend the execution of current thread for a given period.
 * @param dwTimeout  [in] Duration for sleep in milliseconds.
 */
palEXPORT void palSleep( palDWORD dwTimeout );


/******************************************************************************
 * Time
 ******************************************************************************/
```

```
typedef void (*palTimerCallback_t)( palDWORD dwUserId );

/**
 * PAL interface to create a timer.
 * @param pFuncCb  [in] Pointer to a function to call when timer expires.
 * @param pvUser  [in] User data passed to timer callback as a parameter.
 * @param pdwTimerId  [out] Unique Id to use when using timer routines.
 * @return  palTRUE on success, palFALSE otherwise.
 */
palEXPORT palBOOL palCreateTimer( palTimerCallback_t  pFuncCb,
                                              void                *dwUserId,
                                              palDWORD            *pdwTimerId );

/**
 * PAL interface to delete a timer.
 * @param dwTimerId  [in] Unique timer Id returned by palCreateTimer().
 * @return  palTRUE on success, palFALSE otherwise.
 */
palEXPORT palBOOL palDeleteTimer( palDWORD dwTimerId );

/**
 * PAL interface to start a timer.
 * @param dwTimerId  [in] Unique timer Id.
 * @param dwMSec  [in] Periodic timer duration in milliseconds.
 * @return  palTRUE on success, palFALSE otherwise.
 */
palEXPORT palBOOL palStartTimer( palDWORD dwTimerId, palDWORD dwMSec );

/**
 * PAL interface to stop a timer.
 * @param dwTimerId  [in] Unique timer Id.
 * @return  palTRUE on success, palFALSE otherwise.
 */
palEXPORT palBOOL palStopTimer( palDWORD dwTimerId );

/**
 * PAL interface to return system time.
 * @return  32-bit value of the system timer.
 */
palEXPORT palDWORD palGetTime( void );


/*******************************************************************************
 * Mutex
 ******************************************************************************/

/**
 * PAL interface to initialize a mutex object.
 * @param ppvMutex  [out] pointer to pointer to a mutex object.
 * @param szName  [in] unique name for memory segment.
 * @return  palTRUE on success, palFALSE otherwise.
 */
palEXPORT palBOOL palMutexInit( void* *ppvMutex, const palCHAR *szName);

/**
 * PAL interface to destroy a mutex object.
 * @param pvMutex  [in] Pointer to a mutex object.
 */
palEXPORT void palMutexDestroy( void* pvMutex );

/**
 * PAL interface to lock a mutex object.
 * @param pvMutex  [in] Pointer to a mutex object.
 * @return  palTRUE on success, palFALSE otherwise.
 */
palEXPORT palBOOL palMutexLock( void* pvMutex );

/**
 * PAL interface to unlock a mutex object.
 * @param pvMutex  [in] Pointer to a mutex object.
 * @return  palTRUE on success, palFALSE otherwise.
```

 Copyright © 2001 – 2010 S2 Technologies, Inc.

```
 */
palEXPORT palBOOL palMutexUnlock( void* pvMutex );


/******************************************************************************
 * Memory Management
 ******************************************************************************/

/**
 * PAL interface to allocate dynamic memory.
 * @param wSize  [in] Size in bytes of the memory request.
 * @return  Pointer to block of memory allocated on success, NULL otherwise.
 */
palEXPORT void* palMemAlloc( palWORD wSize );

/**
 * PAL interface to free allocated memory.
 * @param pvMem  [in] Address of memory to release.
 * @return  palTRUE on success, palFALSE otherwise.
 */
palEXPORT palBOOL palMemFree( void* pvMem );

/**
 * PAL interface to create/open memory segment. For multi-proc target, shared memory.
 * @param szName  [in] Unique name for memory segment.
 * @param dwSize  [in] Size in bytes of memory segment request.
 * @param pbFirstInst  [out] palTRUE if memory segment was created for first time,
palFALSE otherwise.
 * @return  Starting address of memory segment created or opened on success, NULL
otherwise.
 */
palEXPORT void* palMemSegmentOpen( const palCHAR *szName,
                                              palDWORD       dwSize,
                                              palBOOL       *pbFirstInst );

/**
 * PAL interface to close memory segment.
 * @param pvMem  [in] Starting address of memory segment to close.
 * @return  palTRUE on success, palFALSE otherwise.
 */
palEXPORT palBOOL palMemSegmentClose( void *pvMem );


/******************************************************************************
 * Pool Memory Management
 ******************************************************************************/

palEXPORT void* palPoolMemAlloc( palWORD wSize );

palEXPORT void palPoolMemFree( void *pvMem );


/******************************************************************************
 * Logging
 ******************************************************************************/

typedef enum
{
  palLOG_LEVEL_TRACE,
  palLOG_LEVEL_DEBUG,
  palLOG_LEVEL_INFO,
  palLOG_LEVEL_WARN,
  palLOG_LEVEL_ERROR,
  palLOG_LEVEL_FATAL
} palLogLevel_e;

/**
 * PAL interface to log messages.
 * @param eLevel  [in] Log level.
 * @param szFmt  [in] Formatted string for message.
 * @param ...  [in] (optional) Variable argument list to format szFmt.
```

```
 */
palEXPORT void palLog( palLogLevel_e eLevel, const palCHAR* szFmt, ... );


/*******************************************************************************
 *
 * IO
 *
 ******************************************************************************/

/*******************************************************************************
 * Send
 ******************************************************************************/

typedef palWORD (*palOutPndCallback_t)( void );

typedef palWORD (*palOutRdyCallback_t)( void );

/**
 * PAL interface to query Runtime output queue.
 * @param pFuncCb  [in] Function callback to query Out data pending.
 * @return  the previously registreted callback or palNULL if not such set.
 */
palEXPORT palOutPndCallback_t palOutPndReg( palOutPndCallback_t pFuncCb );

/**
 * PAL interface to indicate transport ready to transmit.
 * @param pFuncCb  [in] Function callback to indicate transport is ready to receive next
packet of data.
 * @return  the previously registreted callback or palNULL if not such set.
 */
palEXPORT palOutRdyCallback_t palOutRdyReg( palOutRdyCallback_t pFuncCb );

/**
 * PAL interface to send data to host.
 * @param pyOutBuffer  [in] Address of the data to transmit.
 * @param wSize  [in] Size in bytes of the data.
 * @return  palTRUE on success, palFALSE otherwise.
 */
palEXPORT palBOOL palOut( const palBYTE *pyOutBuffer, palWORD wSize );


/*******************************************************************************
 * Receive
 ******************************************************************************/

typedef palWORD (*palInDataCallback_t)( const palBYTE *pyInBuffer, palWORD wSize);

/**
 * PAL interface to indicate data ready for Runtime.
 * @param pFuncCb  [in] Function callback routine to call when data is received from
transport.
 * @return  the previously registreted callback or palNULL if not such set.
 */
palEXPORT palInDataCallback_t palInReg( palInDataCallback_t pFuncCb );


#ifdef __cplusplus
}
#endif


#endif /* PAL_H */
```


Copyright © 2001 – 2010 S2 Technologies, Inc.

Appendix B: s2Slap.h

```
/*******************************************************************************
 *
 * FILE NAME: s2Slap.h
 *
 * DESCRIPTION:
 *   Simplified Link-layer Application Protocol
 *
 *
 * -----------------------------------------------------------------------------
 * Copyright 2007 by S2 Technologies, Inc.
 * -----------------------------------------------------------------------------
 *******************************************************************************/

#ifndef S2SLAP_H
#define S2SLAP_H

#ifdef __cplusplus
extern "C"
{
#endif

#include <pal.h>

/*******************************************************************************
 *
 * SLAP Control Characters
 *
 *******************************************************************************/

#define S2_SLAP_PREAMBLE 0x7e
#define S2_SLAP_ESCAPE   0x7d
#define S2_SLAP_XOR      0x20


/*******************************************************************************
 *
 * SLAP Header
 *
 *******************************************************************************/

/* Payload Types */
#define S2_SLAP_TYPE_IBLOCK            1

/* Header Size */
#define S2_SLAP_HEADER_SIZE            6


/*******************************************************************************
 *
 * SLAP Errors
 *
 *******************************************************************************/

/* Error Codes */
#define S2_SLAP_ERROR_BAD_CHECKSUM     1
#define S2_SLAP_ERROR_PREAMBLE_IN_DATA 2
#define S2_SLAP_ERROR_LENGTH           3


/*******************************************************************************
 *
 * SLAP API
 *
 *******************************************************************************/

palBOOL s2slapTxMsgFormat(palBYTE       *pyDest,      /* Destination for Slap Frame */
                          palWORD        wDestSize,   /* Size of Destination Buffer */
                          palWORD       *pwDestSize,  /* Size of Slap Frame         */
```

```
                          const palBYTE *pySrc,      /* Data to Frame              */
                          palWORD        wSrcSize,   /* Size of data               */
                          palBYTE        yType);     /* Payload Type               */


palBOOL s2slapRxMsgExtract(palBYTE          *pyDest,       /* Destination buffer
*/
                           palWORD          wDestMaxSize, /* Size of Dest Buffer
*/
                           const palBYTE    *pySrc,        /* Data from Slap Frame
*/
                           palWORD          wSrcSize);     /* Size of data
*/


/* Returns the length of the SLAP frame including the S2_SLAP_HEADER_SIZE,
   Note: This allows you to read only S2_SLAP_HEADER_SIZE bytes, then use
         this function to determine how many more bytes must be read to
        complete the frame.  pyType and pwChecksum may be NULL if these data
        do not need to be returned.                                       */
palWORD s2slapRxMsgHeader(const palBYTE *pySrc,         /* Header from Slap Frame    */
                          palBYTE       *pyType,        /* Returned message type     */
                          palWORD       *pwChecksum);   /* Returned checksum         */



typedef void (*s2slapDataReadyCb_t)(const palBYTE   *pyDataBuffer, /* Data from Slap
Frame      */
                                    palWORD          wSize,        /* Size of data in
bytes     */
                                    palBYTE          yType);       /* Payload Type
*/

s2slapDataReadyCb_t s2slapRegDataReadyCb(s2slapDataReadyCb_t ptFuncCb);


typedef void (*s2slapErrorCb_t)(palBYTE yError, palWORD wParam1, palWORD wParam2);

s2slapErrorCb_t s2slapRegErrorCb(s2slapErrorCb_t ptFuncCb);


#ifdef __cplusplus
}
#endif

#endif
```

 Copyright © 2001 – 2010 S2 Technologies, Inc.