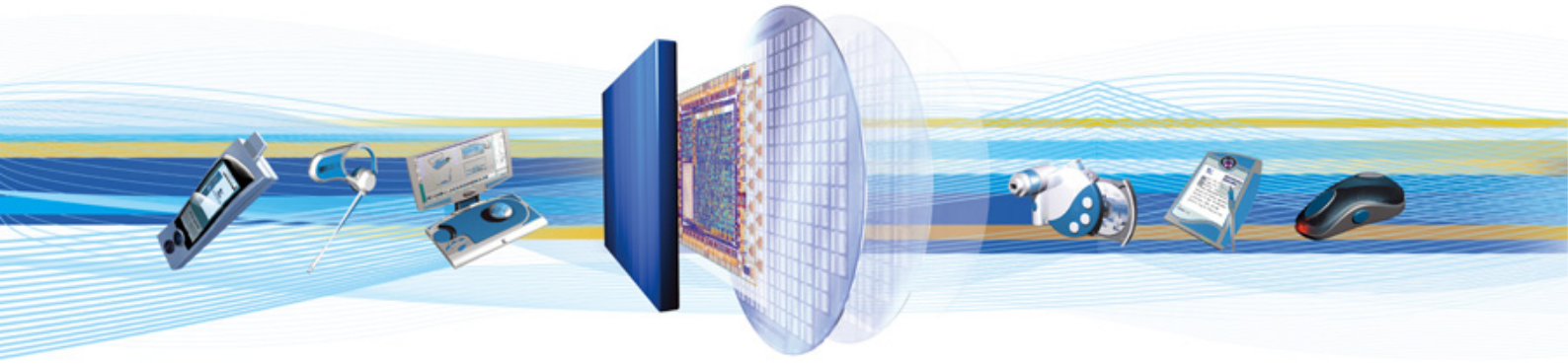
CSR Synergy Framework 3.1.0

Scheduler

API Description

August 2011



Cambridge Silicon Radio Limited

Churchill House
Cambridge Business Park
Cowley Road
Cambridge CB4 0WZ
United Kingdom

Registered in England and Wales 3665875

Tel: +44 (0)1223 692000
Fax: +44 (0)1223 692001
www.csr.com

Contents

List of Tables

1 Introduction

This document describes the CSR Scheduler API. This API is part of the CSR Synergy Framework BSP interface and thus requires platform specific porting. The Scheduler API provides the core services of a virtual operating system, shielding from variations in specific platforms. The Scheduler API consists of the set of API's listed below.

- Task scheduling
- Message forwarding
- Timer handling
- Background interrupts

The following sections provide a detailed description of each of the above APIs. Section 2 describes a set of fundamental types and macros that are required to exist in any BSP. Sections 3-12 present the API's.

Note: For a CSR Synergy Framework architectural overview, please refer to *CSR Synergy Framework Users Guide* [SYN-FRW-USERS-GUIDE].

2 Basic Types

The CSR Synergy Framework provides a set of basic types and macros that are required to exist in any BSP. Their existence is a prerequisite for the Scheduler API as well as all other components in the CSR Synergy Framework and Synergy Technologies. The types and macros must be provided in the file `csr_types.h`. For a reference implementation please refer to e.g.: `$(FW_ROOT)/bsp/ports/pcwin/inc/csr_types.h`.

2.1 Basic Types

The basic type names and their required implementations are shown in Table 1.

Prototype

```
#include "csr_types.h"
```

Type Name	Description
`CsrSize`	Unsigned integer large enough to hold the size of the largest object held in memory (ISO/IEC 9899:1990 7.1.6). This type is the return type of the built-in `sizeof` operator. Usually mapped to `size_t` from `stddef.h`.
`CsrPtrdiff`	Type of the result of subtracting two pointers (ISO/IEC 9899:1990 7.1.6). Usually mapped to `ptrdiff_t` from `stddef.h`.
`CsrUintptr`	Unsigned integer large enough to hold any pointer (ISO/IEC 9899:1999 7.18.1.4). Usually mapped to `uintptr_t` from `stdint.h`. Alternatively can be mapped to `CsrSize`, if `uintptr_t` is not available available or does not conform to ISO/IEC 9899:1999.
`CsrIntptr`	Signed integer large enough to hold any pointer (ISO/IEC 9899:1999 7.18.1.4). Usually mapped to `intptr_t` from `stdint.h`. Alternatively can be mapped to `CsrPtrdiff`, if `inptr_t` is not available or does not conform to ISO/IEC 9899:1999.
`CsrUint8`	Unsigned integer of exactly 8 bit . Usually mapped to `uint8_t` from `stdint.h` or `unsigned char`.
`CsrUint16`	Unsigned integer of exactly 16 bit. Usually mapped to `uint16_t` from `stdint.h` or `unsigned short`.
`CsrUint24`	Unsigned integer of exactly 32 bit. Must be mapped to `CsrUint32`. **DEPRECATED:** use `CsrUint32`
`CsrUint32`	Unsigned integer of exactly 32 bit. Usually mapped to `uint32_t` from `stdint.h` or `unsigned int`. Avoid using `unsigned long int` or `unsigned long` unless `int` is 16bit.
`CsrInt8`	Signed integer of exactly 8 bit. Usually mapped to `int8_t` from `stdint.h`, or `signed char`. Please note that this type must be explicitly signed as `char` may be either signed or unsigned depending on the compiler.
`CsrInt16`	Signed integer of exactly 16 bit. Usually mapped to `int16_t` from `stdint.h` or `short`.

CsrInt32	Signed integer of exactly 32 bit. Usually mapped to int32_t from stdint.h or int. Avoid using long int or long unless int is 16bit.
CsrBool	Data type large enough to store values 0 and 1. Usually mapped to CsrUint8.
CsrCharString	Must be mapped to char. Please note that char must **not** be prefixed with signed or unsigned.
CsrUtf8String	UTF-8 character. Must be mapped to CsrUint8.
CsrUtf16String	UTF-16 character. Must be mapped to CsrUint16.

Table 1: Basic Types

2.2 Optional 64 bit Integer Types

Some Synergy components use 64 bit integer types for certain computations. The following types are optional but may be required for using these modules. Module implementations may test for the CSR_HAVE_64_BIT_INTEGERS symbol to select 64 bit implementations or fall back to 32 bit implementations. Also, the relevant modules may be available in a version that does not depend on the presence of 64 bit integers. Consult the relevant module documentation or the customer support group for information.

Note that the symbol CSR_HAVE_64_BIT_INTEGERS may be defined only if the platform and BSP provides the 64 bit types described in the table below. The define is set in csr_types.h.

Prototype

```
#include "csr_types.h"

#DEFINE CSR_HAVE_64_BIT_INTEGERS
```

Type Name	Description
CsrInt64	Signed integer exactly 64 bits wide. Usually mapped to int64_t from stdint.h or long int or long long.
CsrUint64	Unsigned integer exactly 64 bits wide. Usually mapped to uint64_t from stdint.h or unsigned long int or unsigned long long.

Table 2: Optional 64 bit integer types

2.3 Optional Floating Point Types

Some Synergy components use floating point types for certain computations. The following types are optional but may be required for using these modules. Module implementations may test for the CSR_HAVE_FLOATING_POINT symbol to select floating point implementations or fall back to emulations code. Also, the relevant modules may be available in a version that does not depend on the presence of floating point types. Consult the relevant module documentation or the customer support group for information.

Note that the symbol CSR_HAVE_FLOATING_POINT may be defined only if the platform and BSP provides the 64 bit types described in the table below. The define is set in csr_types.h.

Prototype

```
#include "csr_types.h"

#DEFINE CSR_HAVE_FLOATING_POINT
```

Type Name	Description
CsrFloat	32bit single precision floating point type. Must map to float.
CsrDouble	64bit double precision floating point type. Must map to double.

Table 3: Optional floating point types

2.4 Basic Macros

The basic macros must be defined as follows.

Prototype

```
#include "csr_types.h"

#define TRUE   1 /* if not provided by platform */
#define FALSE  0 /* if not provided by platform */
#define NULL   0 /* if not provided by platform */
```

3 Task Scheduling

3.1 Introduction

The fundamental programming model provided by the CSR Synergy Framework is *task based*. That is, code using the Framework is typically structured as a set of *tasks* passing *messages* to each other. Each task is attached to a *message queue*, which together with a *task init* function, a *task message handler* function, and an optional *task deinit function*, uniquely identifies the task.

The task scheduling API consists of typedefs for the above mentioned message queue and task functions. The API does not specify task initialization or scheduling of tasks. The only assumption made, is that the task scheduling model must be non pre-emptive. That is, the task function implementations must be non-blocking. The task initialization and the task scheduling method are on purpose not specified by the API. These mechanisms are highly implementation specific, and the interface is intentionally left flexible to allow for several possible implementations. The actual implementation may depend on the mechanisms provided by the native platform OS.

The native OS may provide a thread or tasks model that will be used as implementation of the Framework task scheduling. The native OS may provide a queue mechanism that can directly implement the Framework message queues. And it may provide a task mechanism including a task handler function that can directly map the Framework task message handler. Also, the scheduling mechanism, or scheduling loop, itself may be part of the native OS task model.

It is also possible to implement a dedicated scheduling mechanism for the Framework, using none or only some parts of the native OS mechanisms. In this approach the actual implementation of the scheduling mechanism may depend on the OS. E.g. in a pre-emptive multitasking OS a dedicated scheduler implementation may use an infinite loop for the scheduler loop, whereas this may not be an option with other OS types. Whichever implementation method is used, the key point is that the Framework shields its users from these details.

In Section 3.2, the fundamental typedefs constituting the task API are described. Section 4 then presents a reference implementation scheme for a dedicated scheduler implementation.

3.2 Basic Types

In the following, the fundamental types for the task message queue and task functions are described. Furthermore, the required semantics of the task functions are described in detail. The task scheduling API contains no actual function prototypes to be ported.

3.2.1 Message Queue Type

The message queue identifier must be defined in file `csr_sched.h` as an unsigned integer of exactly 16 bit.

Typedef

```
#include "csr_sched.h"

typedef CsrUint16 CsrSchedQid;
```

Description

The unique identifier is used for addressing a single queue throughout the scheduler. It is used as the destination address in the message forwarding API.

3.2.2 Task Functions Type

The task init function, message handler function, and task deinit function all share the following prototype, which must be defined in `csr_sched.h`.

Typedef

```
#include "csr_sched.h"

typedef void (*schedEntryFunction_t)(void **inst);
```

Description

The **task init function** is called for exactly one time before its handler function. The init function will typically allocate and initialize the task instance data parameter `void **inst`. If the task does not need any instance data, the init function can be set to `NULL`.

The **task handler function** must be called by the scheduler, when the task queue contains one or more messages, but is not required to consume any pending messages. If a task leaves messages in its queue the scheduler must reinvoke the task later. This forms a technique of keeping the task "alive". Finally, a task can consume as many messages being available in its queue. The `void **inst` pointer argument contains the task instance data.

The optional **task deinit function** is, if it is defined, called by the scheduler to perform a graceful shutdown of a task. This function must: 1) empty the input message queue and free any allocated memory in the messages and 2) free any instance data that may be allocated.

Parameters

Type	Argument	Description
`void**`	`inst`	This pointer argument contains the task instance data. It is typically allocated and initialized in the task init function. It is passed in every call to the task handler function. The optional task deinit function also takes this parameter and free any allocated instance data.

Returns

None.

4 Task Scheduling Reference

4.1 Introduction

As described in Section 3, the task scheduling interface is entirely implementation specific. Only the types described in Section 3 need to be conformant. There is no specification of interfaces for task initialization or for the scheduling of tasks. These interfaces are intentionally left open to allow for several possible implementations depending on the mechanisms provided by the native platform OS. This section presents a reference interface and implementation scheme for the task initialization and task scheduling.

The reference implementation is for a dedicated scheduler implementation assuming a pre-emptive multitasking native OS. Thus allowing a scheduler loop implementation using an infinite loop. The reference API described is used in all the reference ports provided by a CSR Synergy Framework release.

Note: The reference API is contained in file **`csr_sched_init.h`** in the special include directory `$(BSP_ROOT)/inc/platform`. Interfaces in this folder may be platform dependent and they are not necessarily available with any BSP. Thus, the CSR Synergy Framework does not mandate porting of these interfaces. Source code for the reference implementation is contained in file **`csr_sched.c`** in the `$(BSP_ROOT)/src/coal` directory.

Note: : For details on the CSR Synergy Framework directory structure, please refer to the *CSR Synergy Framework Users Guide* [SYN-FRW-USERS-GUIDE].

Sections 4.2 and 4.3 present a reference for scheduler initialization and de-initialization. Sections 4.4 and 4.5 present a reference for starting and stopping the scheduler. Sections 4.6 and 4.7 present a reference for task initialization and de-initialization. Section 4.8 presents a full example of putting together an application using the Framework task scheduling reference implementation. Finally, Section 4.9 describes a reference scheme for implementing a multi instance scheduler.

4.2 CsrSchedInit

Prototype

```
#include "platform/csr_sched_init.h"

void *CsrSchedInit(CsrUint16 id);
```

Description

This function initializes the scheduler implementation. I.e. it initializes the internal structures used by the scheduler like task queue structures, background interrupts, etc. The function returns the private scheduler instance data (as a void pointer), which must be passed to subsequent scheduler functions. The `id` parameter passed to the function is the scheduler identification used for the routing of internal/external messages. When the function is called, the Framework scheduler reference implementations perform a function callback to the task initialization function `CsrSchedTaskInit()` described in Section 4.6.

Parameters

Type	Argument	Description
`CsrUint16`	`id`	The scheduler identifier used for the routing of internal / external messages.

Returns

The private scheduler instance data as a void pointer.

4.3 CsrSchedDeinit

Prototype

```
#include "platform/csr_sched_init.h"

void CsrSchedDeinit(void *data);
```

Description

Cleanup and free the scheduler instance data.

Parameters

Type	Argument	Description
`void*`	`data`	The scheduler instance data, returned by `CsrInitSched()`

Returns

None.

4.4 CsrSched

Prototype

```
#include "platform/csr_sched_init.h"

void CsrSched(void *data);
```

Description

Start the scheduler. The function will not return until the scheduler has been instructed to exit. All reference implementations enter an infinite loop in which tasks are continuously scheduled in round-robin fashion. This scheme makes use of the assumption of a pre-emptive multitasking native OS. The `data` parameter is the instance returned by `CsrSchedInit()`.

It is possible to implement the scheduler both as a non-blocking function and as a blocking function. The essential consideration when designing the scheduler loop as a blocking function is that it can only be blocked in one place in the scheduler loop. The purpose of this is that the scheduler can then remain blocked until an external event occurs. The external event may e.g. be a background interrupt signalling that data from the low-level transport drivers is available, it may be message received from another native OS task, or it may be a timed event.

Parameters

Type	Argument	Description
`void*`	`data`	The scheduler instance data, returned by `CsrInitSched()`

Returns

None.

4.5 CsrSchedStop

Prototype

```
#include "platform/csr_sched_init.h"

void CsrSchedStop(void);
```

Description

Stop the scheduler. The function will raise a flag causing the scheduler to finish the message- and timer-handling loop and return from the `CsrSched()` function.

Parameters

None.

Returns

None.

4.6 CsrSchedTaskInit

Prototype

```
void CsrSchedTaskInit(void *data);
```

Note: This is a user supplied function. Scheduler reference implementations assume this function is implemented and exposed by the application.

Description

This function sets up all the scheduler tasks functions: task message handlers, and task init- and task deinit-functions. The scheduler reference implementations all have an interface to tie the above task functions together with a task queue id and register this with the scheduler.

Parameters

Type	Argument	Description
`void*`	`data`	The scheduler instance data, returned by `CsrInitSched()`

Returns

None.

4.7 CsrSchedTaskDeinit

Prototype

```
#include "platform/csr_sched_init.h"

void CsrSchedTaskDeinit(void *data);
```

Description

This function calls all task deinit function, if any, and cancels any remaining timers and/or messages. Finally, the scheduler instance itself will be freed using the function. The function must be called after termination of the scheduler loop, i.e. return from `CsrSched()`.

Parameters

Type	Argument	Description
`void*`	`Data`	The scheduler instance data, returned by `CsrInitSched()`

Returns

None.

4.8 Task Example

This section describes example code using the above task scheduling reference API. The example code describes a very simple example of how to declare and register two tasks with the scheduler. It also shows how to start and stop the scheduler.

As mentioned earlier, a task interface consists of the task message queue, the task init function, the task handler function, and the optional task deinit function. The declaration of task interfaces is typically provided in a header file of its own. Table 4 shows the task interface for the simple example – for simplicity the declarations are only shown for one task. Table 5 shows the implementation for the task init and task handler function of this one task.

Notice in Table 4, that the task deinit function is only defined if the symbol `ENABLE_SHUTDOWN` has been defined. The Framework allows this symbol to be configured via its build system. Please refer to the *CSR Synergy Framework Users Guide* [SYN-FRW-USERS-GUIDE] for details of the build system.

```
extern void Task1Init(void **gash);
extern void Task1Handler(void **gash);
#define TASK1_INIT      Task1Init
#define TASK1_HANDLER   Task1Handler
#ifdef ENABLE_SHUTDOWN
extern void Task1Deinit(void **gash);
#define TASK1_DEINIT    Task1Deinit
#else
#define TASK1_DEINIT    NULL
#endif

extern CsrSchedQid TEST_TASK1_IFACEQUEUE;

/* Likewise for TASK2 */
```

Table 4: Example Task Interface `(tasks.h)`

In the example task implementation the first task, in its init function, allocates and sends a message to the second task. In its handler function, it simple gets the message of its queue and frees it.

```
void Task1Init(void **gash)
{
    InstanceData_t *theData;
    CsrUint16 *msg;

    *gash = (void*) CsrPmemAlloc(sizeof(InstanceData_t));
    theData = (InstanceData_t *) *gash;

    msg = (CsrUint16*) CsrPmemAlloc(sizeof(CsrUint16));
    msg= 0;
    CsrSchedMessagePut(TEST_TASK2_IFACEQUEUE, 1, msg);
}

void Task1Handler(void **gash)
{
    InstanceData_t *theData;
    void *receivedmsg;
    CsrUint16 eventClass;
```

```
    theData = (InstanceData_t *) (*gash);

    CsrSchedMessageGet(&eventClass, &receivedmsg);

    CsrPmemFree(receivedmsg);
}
```

Table 5: Example Task Implementation

The application code using the task interface will include the above interface. It will implement the `CsrTaskInit()` function that registers the tasks with the scheduler. Typically, this will be performed in a separate module (tasks.c). Table 6 shows the task registration for the simple example.

The example defines two global task queues with a default initialization. The scheduler implementation will internally assign unique values to the task queue id variables. The `CsrSchedRegisterTask()` function registers the tasks with the scheduler. The final parameter, `0`, of the function indicates the scheduler instance in which the registered task will run. In this example there is only one instance. Section 4.9 describes in more detail the use of multiple scheduler instances.

```
#include "tasks.h"

CsrSchedQid TEST_TASK1_IFACEQUEUE = CSR_SCHED_TASK_ID;
CsrSchedQid TEST_TASK2_IFACEQUEUE = CSR_SCHED_TASK_ID;

void CsrSchedTaskInit(void *data)
{
    CsrSchedRegisterTask(&TEST_TASK1_IFACEQUEUE, TASK1_INIT, TASK1_DEINIT,
        TASK1_HANDLER, "TEST_TASK1", data, 0);
    CsrSchedRegisterTask(&TEST_TASK2_IFACEQUEUE, TASK2_INIT, TASK2_DEINIT,
        TASK2_HANDLER, "TEST_TASK2", data, 0);

}
```

Table 6: Example Task Scheduler Registration

The application code starting the scheduler will typically be started from the `main()` function. See Table 7. The scheduler will be initialized via the `CsrSchedInit()` call which initializes scheduler instance data and calls back to the `CsrSchedTaskInit()` function. Notice that the scheduler is initialized with scheduler instance number `0`. This matches the scheduler instance parameter of the `CsrSchedRegisterTask()` calls. Once this has happened, the scheduler loop can be started with the `CsrSched()` call. This loop will only be stopped if the `CsrSchedStop()` function is called. This will typically be done from somewhere in the application code when a graceful shutdown is required. Upon shutdown, the deinit function for the tasks `CsrSchedTaskDeinit()` and the scheduler itself `CsrDeinitSched()` will be called.

```
int main(int argc, char *argv[])
{
    void *schedInstance;
    …
    schedInstance = CsrSchedInit(0);
    …
    CsrSched(schedInstance);
    …
    /* Graceful shutdown */
#ifdef ENABLE_SHUTDOWN
    CsrSchedTaskDeinit(schedInstance);
    CsrSchedDeinit(schedInstance);
#endif
    return 0;
}
```

Table 7: Example Task Scheduler Start

4.9 Multiple Scheduler Instances

In some setups, it may e.g. be necessary to run the scheduler in multiple threads. This requires the scheduler implementation to be able to handle multiple instances. This section describes an implementation scheme for multiple scheduler instances. In the previous section, it was briefly described how the reference interface allows for scheduler initialization and task registration with a scheduler instance identifier. This section describes in more detail how the scheduler implementation may use this instance information. The CSR Synergy Framework bdb2 reference port contains a full implementation of a multi instance scheduler allowing multiple threads to run separate instances. Source code for the bdb2 reference implementation is contained in file `$(BSP_ROOT)/src/sched/nucleus/csr_sched.c`.

Scheduler Identifier

In order to differentiate between multiple schedulers or multiple threads it is necessary to know *where* a given task is run, i.e. in which scheduler it is run. To accomplish this a *scheduler identification* is used, which is the id that is passed as a parameter to `CsrSchedInit()` and to the `CsrSchedRegisterTask()` function.

Each scheduler then knows it's own identifier, and through `CsrSchedRegisterTask()` the identifier of all other tasks. This also means that the scheduler has knowledge of which tasks are local and which are run in a different context. In the reference implementation the actual scheduler identifier must be a number between 0 (zero) and 7. Note that this range corresponds to the bits 0, 1 and 2 (least significant).

Unique Task Queue

For the message routing to work, tasks must be uniquely identified across all schedulers. This is accomplished by letting the scheduler assign the actual task queue id during the task setup phase. The task queue id is assigned by the scheduler implementation as follows:

```
QUEUE = taskCount | (id << 12);
taskCount++;
```

where

- `QUEUE` is the task queue id
- `taskCount` is a local scheduler instance variable that is initialised to 0 (zero) when `CsrInitSched()` is called and incremented each time a new task is initialised via `CsrSchedRegisterTask()`
- `id` is the scheduler identifier passed as a parameter to the `CsrSchedRegisterTask()` function.

As the `id` passed to `CsrSchedRegisterTask()` does not change between the different schedulers, the identifier is globally unique even across multiple schedulers compiled for different setups. In other words, the scheduler identifier is incorporated into the queue number as bit 12, 13 and 14 (least significant with index start at bit zero).

Obtaining Routing Information for Messages

When a scheduler receives a message at a particular queue, it is possible to extract the scheduler identifier using the following scheme:

```
schedId = (queue & 0x7000) >> 12;
```

Where

- `schedId` is the scheduler identifier.
- `queue` is the queue for which the message is intended.
- `0x7000` is the mask used for filtering out all other bits than index 12, 13 and 14
- `12` is the number of indices that the number must be bit-shifted right to obtain the scheduler identifier from bit 12, 13 and 14.

With the `schedId` number at hand, the scheduler can then compare this to it's own local scheduler identifier and either put the message on a local scheduler queue should the two identifiers match – or otherwise sent the message to an external scheduler.

5 Message Forwarding

5.1 Introduction

As mentioned before, each task has two main elements a message queue and a message handler function. The message queue accumulates messages, and the message handler consumes messages and acts upon them. This section describes the function prototypes of the message forwarding API.

5.2 Basic Types

The message forwarding API uses a single message identifier type.

5.2.1 Message Identifier Type

The message identifier type must be defined in file `csr_sched.h` as an unsigned integer of exactly 32 bit.

Typedef

```
#include "csr_sched.h"

typedef CsrUint32 CsrSchedMsgId;
```

Description

A message identifier.

5.2.2 External Send Macro

Macro

```
#include "csr_msg_transport.h"

#define CsrMsgTransport <CsrSchedMessagePut prototype>
```

Description

To separate the communication of external and internal messages, an *external message send* macro exists. The macro must map to a function with the same prototype as `CsrSchedMessagePut`. The macro is used for sending messages from external tasks (e.g. from native OS tasks) into the scheduler. This is required in the typical porting setup, where the scheduler tasks runs in another OS task than the application. If a port is used where all tasks runs in the scheduler, this macro can be mapped directly to `CsrSchedMessagePut`. This is the implementation in all the Framework reference ports.

This macro is used by Synergy Technologies to implement generic message sending functions, which shield users from knowledge of whether messages are sent internally between scheduler tasks, between multiple scheduler instances, or out of the scheduler.

5.3 CsrSchedMessagePut

Prototype

```
#include "csr_sched.h"

void CsrSchedMessagePut(CsrQid q, CsrUint16 mi, void *mv);
```

Description

Sends a message, consisting of the integer `mi` and the void pointer `mv`, to the message queue `q`. The `mi` and `mv` are supposed neither to be inspected nor changed by the scheduler. The task that owns q is expected to make sense of the values. The pointer `mv` can be `NULL`, however, typically it will point at data, i.e. a chunk of `CsrPmemAlloc()`ed memory. Tasks should normally obey the convention that when a message built with `CsrPmemAlloc()`ed memory is given to `CsrSchedMessagePut()`, then ownership of the memory is passed on to the scheduler - and eventually to the recipient task. I.e., the receiver of the message will be expected to `CsrPmemFree()` the message storage.

Parameters

Type	Argument	Description
`CsrSchedQid`	`q`	The destination queue.
`CsrUint16`	`mi`	Message integer part. Typically, the message type or event class.
`void*`	`mv`	Message pointer part. Typically, the message payload data.

Returns

void.

5.4 CsrSchedMessageGet

Prototype

```
#include "csr_sched.h"

CsrBool CsrSchedMessageGet(CsrUint16 *pmi, void **pmv);
```

Description

Obtains a message, consisting of the integer pointed to by `pmi` and the void pointer pointed to by `pmv`, from the message queue belonging to the task calling the function. If a message is taken from the queue, then `pmi` and `pmv` are set to the `mi` and `mv` passed to `CsrSchedMessagePut()`.

Parameters

Type	Argument	Description
`CsrUint16*`	`Pmi`	Pointer to message integer part.
`Void**`	`Pmv`	Pointer to message pointer part.

Returns

TRUE if a message has been obtained from the queue; FALSE otherwise.

5.5 CsrSchedTaskQueueGet

Prototype

```
#include "csr_sched.h"

CsrQid CsrSchedTaskQueueGet(void);
```

Description

Used by a task to obtain the queue identifier by which it is identified to the scheduler implementation. Typically, used by code where the same task message handler function is used with multiple queues. The queue identifier is needed e.g. when a task needs to inform another task of its queue ID such that it can send back messages.

Parameters

None.

Returns

Return the queue identifier for the currently running task, or 0xFFFF if not available.

6 Timer Handling

6.1 Introduction

The timer handling interface contains services for generating timed events in the scheduler, i.e. calling functions after a given time. This section describes the function prototypes of the timer handling API.

6.2 Basic Macros

A set of basic time macros must be defined in `csr_sched.h` as follows.

Prototype

```
#include "csr_sched.h"

#define CSR_SCHED_TIME_MAX      ((CsrTime) 0xFFFFFFFF)
#define CSR_SCHED_MILLISECOND   ((CsrTime)(1000))
#define CSR_SCHED_SECOND        ((CsrTime)(1000 * CSR_SCHED_MILLISECOND))
#define CSR_SCHED_MINUTE        ((CsrTime)(60 * CSR_SCHED_SECOND))
```

6.3 CsrSchedTimerSet

This function is used for directing the scheduler to call a given function after a specified amount of time.

Prototype

```
#include "csr_sched.h"

CsrSchedTid CsrSchedTimerSet(CsrTime delay,
                      void (*fn)(CsrUint16 mi, void *mv),
                      CsrUint16 fniarg,
                       void *fnvarg);
```

Description

Causes the function `fn()` to be called with the arguments `fniarg` and `fnvarg` after `delay` has elapsed. The function does nothing with `fniarg` and `fnvarg` except pass them on to `fn()`.

> **Note:** The function will be called at or after `delay`; the actual delay will depend on the timing behaviour of the scheduler's tasks.

Parameters

Type	Argument	Description
`CsrTime`	`delay`	Number of microseconds to delay before calling `fn`.
`void (*f)(CsrUint16, void *)`	`fn`	Function to call.
`CsrUint16`	`fniarg`	Argument size.
`void*`	`*fnvarg`	Argument pointer.

Returns

A timed event identifier, can be used in `CsrSchedTimerCancel()`.

6.4 CsrSchedTimerCancel

Prototype

```
#include "csr_sched.h"
CsrBool CsrSchedTimerCancel(CsrSchedTid eventid, CsrUint16 *pmi, void
**pmv);
```

Description

The function can be used for attempting to prevent delivery of a message sent with `CsrSchedTimerSet()`. Remember the caller must again dispose of any `CsrPmemAlloc()`ed memory thoughtfully.

Parameters

Type	Argument	Description
`CsrSchedTid`	`eventid`	The timed event id to cancel. Passing CSR_SCHED_TID_INVALID is legal and has no effect. In this case the function returns FALSE.
`CsrUint16*`	`pmi`	Pointer to argument size of timed event.
`void**`	`pmv`	Pointer to argument pointer of timed event.

Returns

TRUE if cancelled, FALSE if the event has already occurred.

7 Background Interrupts

7.1 Introduction

This background interrupt interface contains a mechanism to signal the execution of certain processing at the next available time when no tasks are running. Background interrupts are e.g. used by transport drivers running in a separate thread to signal the background task (the scheduler) that data available. This will avoid that the background task needs to poll the buffers. A background interrupt needs to be registered with the scheduler. The registration must be performed from within the function that makes use of the background interrupt function. The registration defines the interrupt number and the function to be activated.

The background interrupt API must be provided in the file `csr_sched.h`.

7.2 Basic Types

The background interrupt API uses the basic types described in this section. These basic types must all be provided by the file `csr_sched.h`.

7.2.1 Background Interrupt Type

Typedef

```
#include "csr_sched.h"

typedef CsrUint16 CsrSchedBgint;
```

Description

Background interrupt type.

7.2.2 Background Interrupt Handler Type

Typedef

```
#include "csr_sched.h"

typedef void (*CsrSchedBgintHandler)(void *);
```

Description

Background interrupt handler function type.

7.3 CsrSchedBgintRegister

Prototype

```
#include "csr_sched.h"
CsrBgint CsrSchedBgintRegister(CsrSchedBgintHandler cb,
                               void *context,
                               const CsrCharString *id);
```

Description

Register a background interrupt handler function with the scheduler. When `CsrSchedBgintSet()` is called from the foreground (e.g. an interrupt routine) the registered function is called. If `cb` is null then the interrupt is effectively disabled. If no bgints are available, `CSR_SCHED_BGINT_INVALID` is returned, otherwise a `CsrSchedBgint` value is returned to be used in subsequent calls to `CsrSchedBgint()`. The `id` parameter is a possibly `NULL` identifier used for logging purposes only.

Parameters

Type	Argument	Description
`CsrSchedBgintHandler`	`cb`	The interrupt handler function.
`void*`	`context`	User data.
`const CharString*`	`id`	Identification for logging.

Returns

Background interrupt identifier – `CSR_SCHED_BGINT_INVALID` if no available bgints.

7.4 CsrSchedBgintUnregister

Prototype

```
#include "csr_sched.h"
void CsrSchedBgintUnregister(CsrSchedBgint irq);
```

Description

Unregister a background interrupt handler function.

Parameters

Type	Argument	Description
`CsrSchedBgint`	`irq`	Background interrupt identifier previously obtained from `CsrSchedBgintRegister()`.

Returns

None.

7.5 CsrSchedBgintSet

Prototype

```
#include "csr_sched.h"
void CsrSchedBgintSet(CsrSchedBgint irq);
```

Description

Sets a background interrupt. Signals the scheduler to call the function, which is registered with the background identifier `irq`.

Parameters

Type	Argument	Description
`CsrSchedBgint`	`irq`	Background interrupt identifier previously obtained from `CsrSchedBgintRegister()`.

Returns

None.

8 Document References

[SYN-FRW-USERS-GUIDE]	CSR Synergy Framework Users Guide. Doc. gu-0001-users_guide

Terms and Definitions

Abbreviation	Explanation
CSR	Cambridge Silicon Radio

Document History

Revision	Date	History
1	14 JUL 09	Initial version
2	19 OCT 09	Rename to Scheduler API and move Time, Panic, and Exception APIs
3	23 NOV 09	Updated with new CsrSched namespacing + remove pmalloc (separate Pmem API)
4	30 NOV 09	Ready for release 2.0.0
5	20 APR 10	Ready for release 2.1.0
6	OCT 10	Ready for release 2.2.0
7	DEC 10	Ready for release 3.0.0
8	Aug 11	Ready for release 3.1.0

TradeMarks, Patents and Licences

Unless otherwise stated, words and logos marked with ™ or ® are trademarks registered or owned by CSR plc or its affiliates. Bluetooth® and the Bluetooth logos are trademarks owned by Bluetooth SIG, Inc. and licensed to CSR. Other products, services and names used in this document may have been trademarked by their respective owners.

The publication of this information does not imply that any licence is granted under any patent or other rights owned by CSR plc.

CSR reserves the right to make technical changes to its products as part of its development programme.

While every care has been taken to ensure the accuracy of the contents of this document, CSR cannot accept responsibility for any errors.

No statements or representations in this document are to be construed as advertising, marketing, or offering for sale in the United States imported covered products subject to the Cease and Desist Order issued by the U.S. International Trade Commission in its Investigation No. 337-TA-602. Such products include SiRFstarIII™ chips that operate with SiRF software that supports SiRFInstantFix™, and/or SiRFLoc® servers, or contains SyncFreeNav functionality.

Life Support Policy and Use in Safety-critical Compliance

CSR's products are not authorised for use in life-support or safety-critical applications. Use in such applications is done at the sole discretion of the customer. CSR will not warrant the use of its devices in such applications.

Performance and Conformance

Refer to www.csrsupport.com for compliance and conformance to standards information.