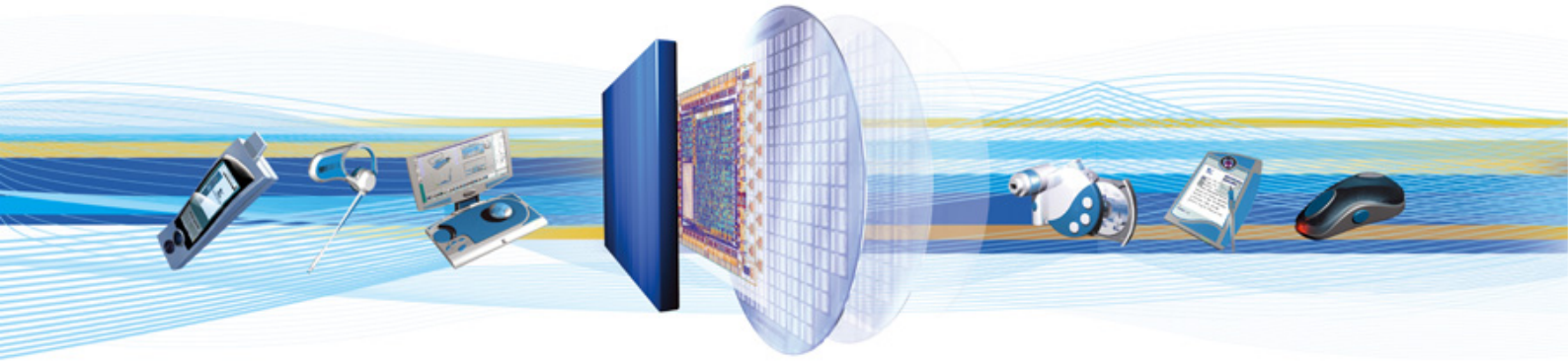
CSR Synergy Framework 3.1.2

TM – Transport Manager

API Description

January 2012



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 Figures

List of Tables

1 Introduction

This document describes the API for the CSR Synergy Transport Manager (TM). The TM API provides an interface to initialise and activate the host interface to a BlueCore® chip, as well as an optional interface to monitor the status of the chip/host interface. Section 2 presents the basic part of the TM API and Section 3 presents the optional monitoring API, named the TM Chip Manager API.

The document is intended for application developers as well as developers of Synergy technologies.

2 TM API

The basic service provided by the TM is that of initialising and activating the host interface to a BlueCore® chip. As part of this service, the TM also controls the download of patches and configuration data based on the BlueCore® firmware build id. This download mechanism is generally referred to as 'bootstrapping' the BlueCore®. The TM API specifies an interface by which an application can provide a bootstrap sequence to be applied during initialisation.

2.1 Transport Initialisation

In order to use the TM API it must be initialised with the specific host transport used to interface to the BlueCore® chip. The TM API supports the following list of BlueCore® host transports:

- BCSP (UART)
- H4DS (UART)
- H4I (H4+ variant, UART)
- USB
- TYPEA (SDIO)
- HCI Socket (Linux only)

Each of the above host transports has a specific initialisation function. The TM API is implemented as a task in the Synergy Framework Scheduler and the TM API is initialised to use a specific transport by providing the transport initialisation function as entry function for the TM task. The following describes the initialisation process used for all transports. Additional requirements for the HCI Socket transport initialisation and build requirements are described in Section 2.1.1.

The TM task initialisation API including macros for the transport initialisation functions resides in the TM task interface file: csr_tm_bluecore_task.h. It defines the following macros for the transport initialisation functions:

```
#define CSR_TM_BLUECORE_BCSP_INIT    CsrTmBlueCoreBcspInit

#define CSR_TM_BLUECORE_H4DS_INIT    CsrTmBlueCoreH4dsInit

#define CSR_TM_BLUECORE_H4I_INIT     CsrTmBlueCoreH4iInit

#define CSR_TM_BLUECORE_USB_INIT     CsrTmBlueCoreUsbInit

#define CSR_TM_BLUECORE_TYPE_A_INIT CsrTmBlueCoreTypeAInit

#define CSR_TM_BLUECORE_HCI_SOCKET_INIT CsrTmBlueCoreHciSocketInit
```

The following illustrates how to initialise the TM API to use the BCSP transport. Applications must initialise the TM task as follows:

```
#include "csr_tm_bluecore_task.h"

CsrSchedRegisterTask(&CSR_TM_BLUECORE_IFACEQUEUE, CSR_TM_BLUECORE_BCSP_INIT,
    CSR_TM_BLUECORE_DEINIT, CSR_TM_BLUECORE_HANDLER, "CSR_TM_BLUECORE", data, 0);
```

Please refer to *CSR Synergy Framework Scheduler API Description* [SYN-FRW-SCHED-API] for a detailed description of the interfaces to register tasks with the scheduler.

The UART based transports (BCSP, H4DS, H4I) have an additional initialisation requirement. These transports must all be passed a handle to the specific UART used. The TM has a function based sub-API for this, for each of the UART based transports. The following illustrates the initialisation API for BCSP:

```
#include "csr_tm_bluecore_bcsp.h"

void CsrTmBlueCoreRegisterUartHandleBcsp(void *handle);
```

The above initialisation must occur prior to initialising and starting the scheduler.

Note that the transport initialisation described in this section simply sets up the TM to use a specific transport and in addition, for UART based protocols, it binds the transport protocol together with a specific UART handle. Neither of these will actually activate the transport.

2.1.1 HCI Socket Transport (Linux only)

The HCI Socket transport is Linux specific. It has an additional initialisation interface allowing applications to specify the device name (e.g. hci0) of the hci device to bind to when starting. The interface is as follows:

```
include "platform/csr_hci_socket_init.h"

void CsrTransportHciSocketConfigure(const char *device);
```

To build the HCI Socket transport requires header files from the libbluetooth-dev package. Using the apt –get tool these can be installed by:

```
> sudo apt-get install libbluetooth-dev
```

Note: No additional libraries are needed for linking since the HCI Socket transport only uses the Linux socket interface.

To use the HCI Socket transport, the Linux kernel must be configured with Bluetooth support and with driver support for the underlying HCI transport. Currently, only USB is supported as the underlying driver. Thus, the kernel must be configured (.config) with:

```
CONFIG_BT

CONFIG_BT_HCIBTUSB
```

Depending on the used BlueZ version, it may be needed to patch the BlueZ component. A patch diff file can be downloaded from CSR Support for a limited set of Linux versions. Please contact CSR for Linux distributions not included in the patch file.

2.2 Transport Activation and Deactivation

To activate the transport the TM API provides the CSR_TM_BLUECORE_ACTIVATE_TRANSPORT_REQ message. Upon this request the TM will initialize the host transport interface, run any supplied bootstrap sequence to completion, and when the transport has been activated and the host to BlueCore® communication is established, the activating entity will receive a CSR_TM_BLUECORE_ACTIVATE_TRANSPORT_CFM.

Note, that when using the Synergy Bluetooth technology, applications are not required to activate the transport. The Bluetooth technology component will do this. The application is only required to initialize the host transport, as described in the previous section, and provide the bootstrap sequence. However, the TM accepts multiple activation requests and all requesting entities will receive a confirm when the transport has started. Figure 1 illustrates the activation sequence for an application. To enhance readability the TM signals have been abbreviated by changing the CSR_TM_BLUECORE signal prefix to CSR_TM.

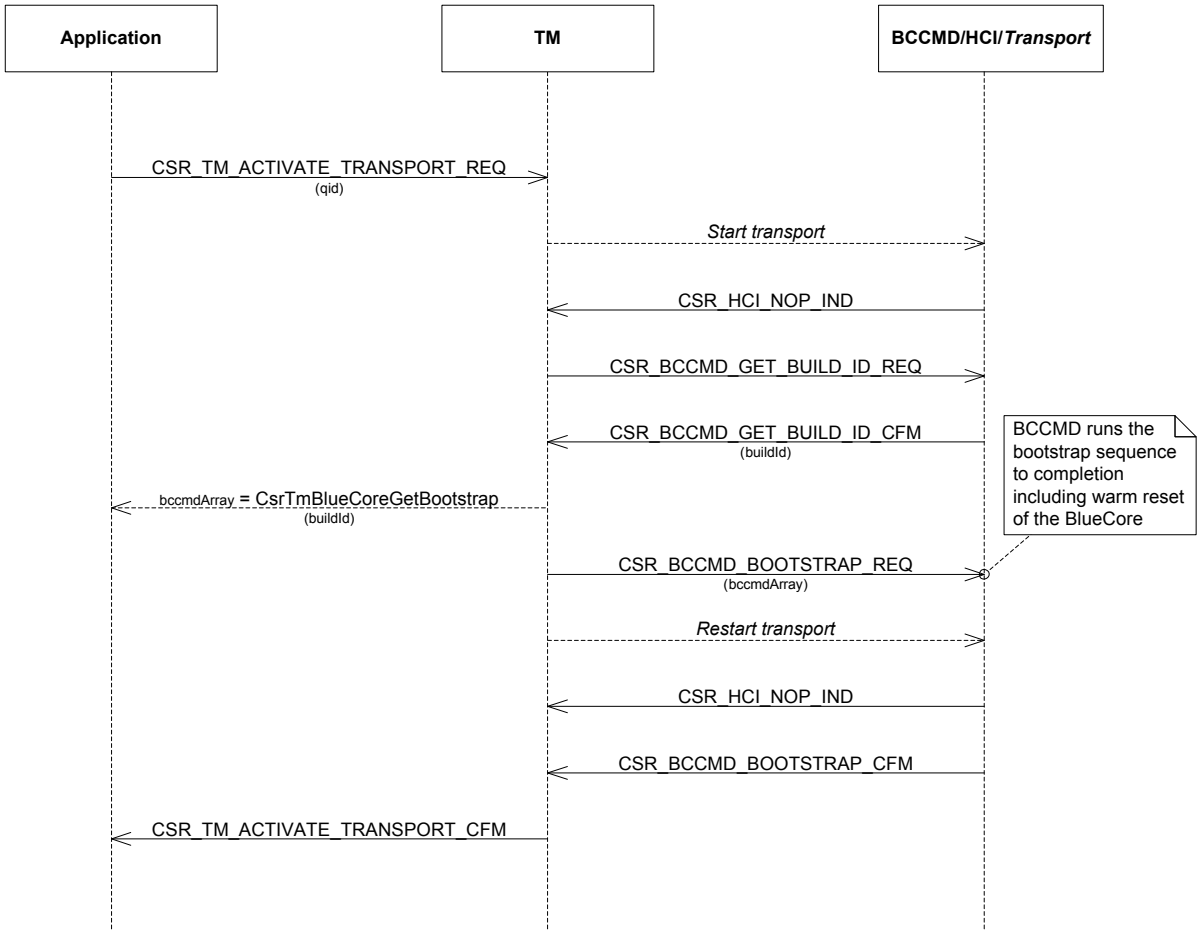


Figure 1: Transport Activation

2.2.1 Application Controlled BlueCore Activation

In the traditional setup, the BlueCore host software is automatically started as soon as the Synergy Scheduler is started, and runs until the scheduler is shut down. If the integration for a particular platform demands complete control of the activation and deactivation of the BlueCore host software, without shutting down the scheduler (which may need to be kept running due to other components), it is possible to enable the CSR_BLUECORE_ONOFF compile time option in the csr_usr_config.h configuration header file.

When this option is enabled, the CSR_TM_BLUECORE_ACTIVATE_TRANSPORT_REQ message will not be sent by any stack component, and the stack will remain idle until the application chooses to do so. It also enables the CSR_TM_BLUECORE_DEACTIVATE_TRANSPORT_REQ message, allowing the application to deactivate the BlueCore host software. It is important that only one entity control this.

To enable other components to take part in the activation/deactivation, an additional registration interface has been added. This allows any component to register itself as a delegate and receive notification when the BlueCore is activated and deactivated. The delegates are required to respond to (some of) these notification, and thus have the ability to stall the activation and deactivation to perform necessary internal processing before the activation/deactivation can complete.

Figure 2

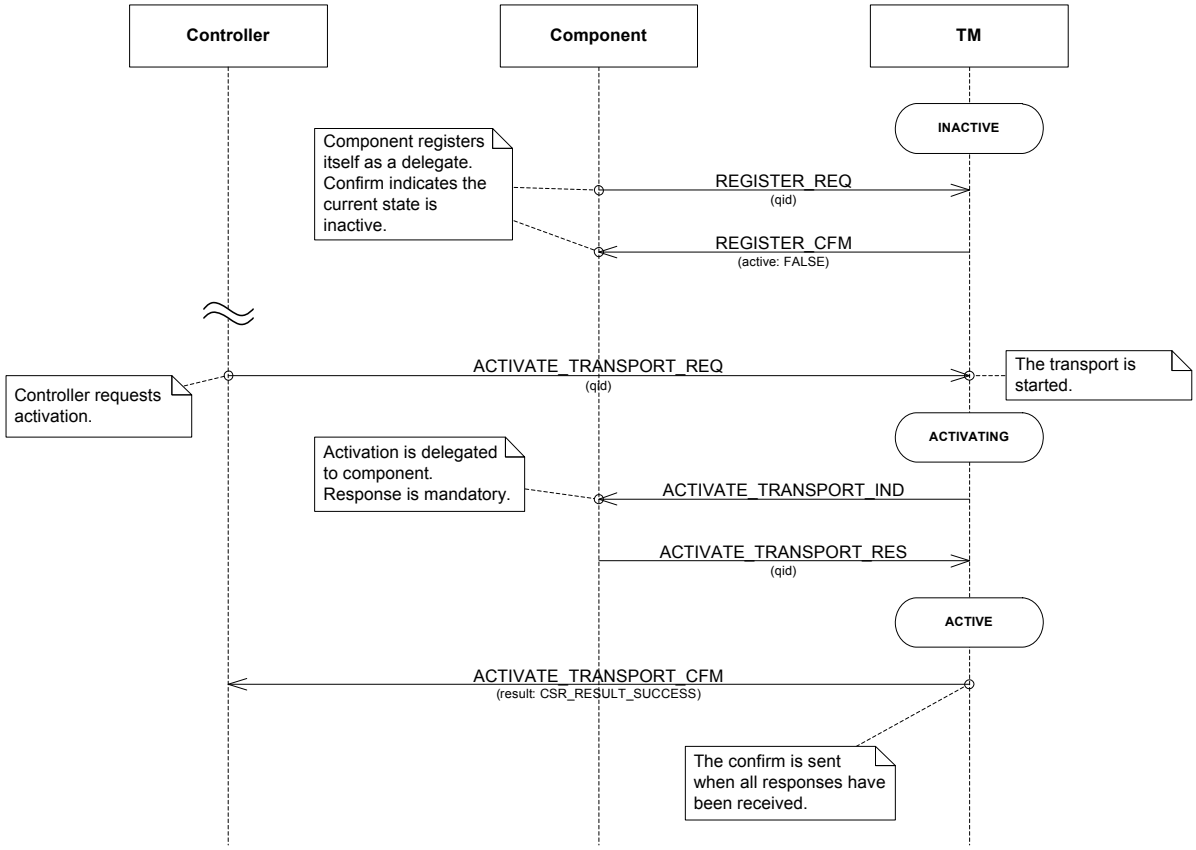


Figure 2: Application Controlled Transport Activation

Figure 2 shows the message sequence for activation when CSR_BLUECORE_ONOFF is defined (The CSR_TM_BLUECORE_ prefix has been omitted for clarity). The middle component is registered as a delegate. As the final step in the activation, it receives an indication, which tells the component that it is now possible to communicate with the BlueCore and that it should allow handling of requests from other components. When all delegates have responded to the indications they have received, the TM will send the CSR_TM_BLUECORE_ACTIVATE_TRANSPORT_CFM to the controlling application, letting it know that the communication with the BlueCore has been established and all stack components are ready to receive requests.

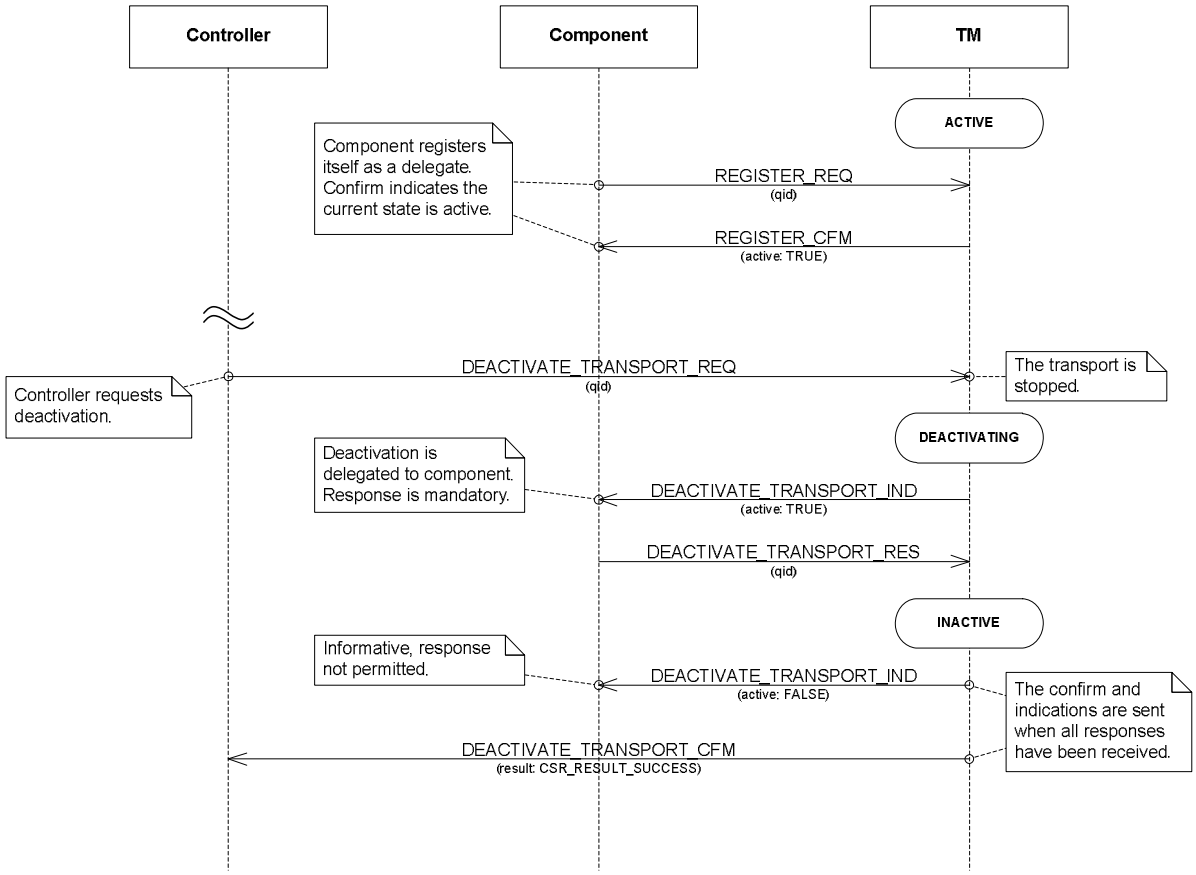


Figure 3: Application Controlled Transport Deactivation

Figure 3 shows the corresponding deactivation sequence when CSR_BLUECORE_ONOFF is defined. The middle component is again registered as a delegate. When the controlling application requests deactivation, the registered delegates receive an indication. Upon reception of this message, the delegate should enter a state where no messages are sent to other tasks (with certain exceptions, see below). When this state has been entered, the delegate responds, to acknowledge this. When all delegates have responded, all delegates will receive a second indication from the TM. Upon reception of this indication, the delegate knows that no further messages will be received from other delegates, and it can complete its cleanup and discard all state relating to the communication with the BlueCore, thereby ensuring that when the BlueCore is again activated, no stale messages and no outdated state will linger and interfere with the following activation cycle.

The general rule is that while a delegate is in the deactivating state (between the first and second deactivation indication), it should discard all incoming messages and not send out any messages to other tasks as it waits for the second deactivation indication. However, certain services are persistent across BlueCore activation and deactivation, and messages relating to these services should not be discarded. This is the case for all messages on the TM itself, because all of its services are persistent across BlueCore activation/deactivation. For example, it is possible to register and unregister as a delegate at any time. Please see the documentation for each task for information about which services are persistent.

One specific case needs to be defined specifically. When a delegate unregisters, it remains registered until it has received a confirm message, it is still required to respond to any activation or deactivation indications as appropriate, until it receives this confirm message. Failure to conform to this rule will result in the activation/deactivation stalling indefinitely, as the TM will wait for the delegate to respond.

2.2.2 CSR_TM_BLUECORE_ACTIVATE_TRANSPORT

The TM API provides the following function for constructing and sending the transport activation message:

```
#include "csr_tm_bluecore_lib.h"

void CsrTmBlueCoreActivateTransportReqSend(CsrSchedQid phandle);
```

The arguments to the function are shown in Table 1.

Type	Parameter	Description

Type	Parameter	Description
CsrSchedQid	phandle	The identity of the calling task

Table 1: CsrTmBlueCoreActivateTransportReqSend

The parameters of the CSR_TM_BLUECORE_ACTIVATE_TRANSPORT_CFM primitive are shown in Table 2

Type	Parameter	Description
CsrTmBlueCorePrim	type	Signal identity – CSR_TM_BLUECORE_ACTIVATE_TRANSPORT_CFM
CsrResult	result	The result of the activation

Table 2: CSR_TM_BLUECORE_ACTIVATE_TRANSPORT_CFM

If CSR_BLUECORE_ONOFF is defined, the CSR_TM_BLUECORE_ACTIVATE_TRANSPORT_IND will be sent to delegates during activation. This message has no parameters (except the signal identity carried by all messages), and the delegate must unconditionally respond using the following function:

```
#include "csr_tm_bluecore_lib.h"

void CsrTmBlueCoreActivateTransportResSend(CsrSchedQid phandle);
```

The arguments to the function are shown in Table 3.

Type	Parameter	Description
CsrSchedQid	phandle	The identity of the calling task

Table 3: CsrTmBlueCoreActivateTransportResSend

2.2.3 CSR_TM_BLUECORE_DEACTIVATE_TRANSPORT

The CSR_TM_BLUECORE_DEACTIVATE_TRANSPORT_REQ message shall only be sent if CSR_BLUECORE_ONOFF is defined. It is sent by the controlling application to request deactivation of the BlueCore. The following function is used for constructing and sending the transport deactivation message:

```
#include "csr_tm_bluecore_lib.h"

void CsrTmBlueCoreDeactivateTransportReqSend(CsrSchedQid phandle);
```

The arguments to the function are shown in Table 4.

Type	Parameter	Description
CsrSchedQid	phandle	The identity of the calling task

Table 4: CsrTmBlueCoreDeactivateTransportReqSend

The parameters of the CSR_TM_BLUECORE_DEACTIVATE_TRANSPORT_CFM primitive are shown in Table 5

Type	Parameter	Description
CsrTmBlueCorePrim	type	Signal identity – CSR_TM_BLUECORE_ACTIVATE_TRANSPORT_CFM
CsrResult	result	The result of the deactivation

Table 5: CSR_TM_BLUECORE_DEACTIVATE_TRANSPORT_CFM

If CSR_BLUECORE_ONOFF is defined, the CSR_TM_BLUECORE_DEACTIVATE_TRANSPORT_IND will be sent to delegates twice during deactivation.

If the communication with the BlueCore chip fails for some reason, the controlling application will be notified by a CSR_TM_BLUECORE_DEACTIVATE_TRANSPORT_IND. The controlling application shall never respond to this with a CSR_TM_BLUECORE_DEACTIVATE_TRANSPORT_RES. Instead it should perform a normal deactivation cycle (by sending the CSR_TM_BLUECORE_DEACTIVATE_TRANSPORT_REQ and waiting for the confirm message) followed by a normal activation cycle (see previous sections about activation and deactivation).

The parameters of the CSR_TM_BLUECORE_DEACTIVATE_TRANSPORT_IND primitive are shown in Table 6

Type	Parameter	Description

Type	Parameter	Description
CsrTmBlueCorePrim	type	Signal identity – CSR_TM_BLUECORE_ACTIVATE_TRANSPORT_IND
CsrBool	active	For delegates: TRUE for the first message which requires a response; FALSE for the second message to which the delegate shall not respond. For controlling application: Always TRUE. The controlling application shall never send a response to this message.

Table 6: CSR_TM_BLUECORE_DEACTIVATE_TRANSPORT_IND

2.2.4 CSR_TM_BLUECORE_REGISTER

The CSR_TM_BLUECORE_REGISTER_REQ message is sent by any task to register itself as a delegate. This message shall only be sent if CSR_BLUECORE_ONOFF is defined.

The following function is used for constructing and sending the register request message:

```
#include "csr_tm_bluecore_lib.h"

void CsrTmBlueCoreRegisterReqSend(CsrSchedQid phandle);
```

The arguments to the function are shown in Table 7.

Type	Parameter	Description
CsrSchedQid	phandle	The identity of the calling task

Table 7: CsrTmBlueCoreRegisterReqSend

After processing the request, the TM will send a CSR_TM_BLUECORE_REGISTER_CFM. The parameters of the CSR_TM_BLUECORE_REGISTER_CFM primitive are shown in Table 8.

Type	Parameter	Description
CsrTmBlueCorePrim	type	Signal identity – CSR_TM_BLUECORE_REGISTER_CFM
CsrBool	active	TRUE if the BlueCore has already been activated, FALSE otherwise.

Table 8: CSR_TM_BLUECORE_REGISTER_CFM

If a register request is received during activation or deactivation, the processing will be postponed until the activation/deactivation is complete. The “active” parameter will indicate whether the BlueCore is already activated.

If a CSR_TM_BLUECORE_REGISTER_REQ is sent in the task initialisation function, it is guaranteed by design that the “active” parameter of the CSR_TM_BLUECORE_REGISTER_CFM will be FALSE, so delegates can be designed around this assumption if they issue the CSR_TM_BLUECORE_REGISTER_REQ in the task initialisation function.

2.2.5 CSR_TM_BLUECORE_UNREGISTER

The CSR_TM_BLUECORE_UNREGISTER_REQ message is sent by a delegate to unregister itself so that it is no longer a delegate. This message shall only be sent if CSR_BLUECORE_ONOFF is defined.

The following function is used for constructing and sending the unregister request message:

```
#include "csr_tm_bluecore_lib.h"

void CsrTmBlueCoreUnregisterReqSend(CsrSchedQid phandle);
```

The arguments to the function are shown in Table 9.

Type	Parameter	Description
CsrSchedQid	phandle	The identity of the calling task

Table 9: CsrTmBlueCoreUnregisterReqSend

The TM will send a CSR_TM_BLUECORE_UNREGISTER_CFM, which contains no parameters to confirm the request. If an unregister request is received during activation or deactivation, the processing will be postponed until the activation/deactivation is complete. Note that the delegate should not consider itself unregistered until the confirm has been received. Consequently, it shall continue to respond to activation and deactivation indications as appropriate until it receives the confirm for the unregister request. Failure to conform to this will result in the activation or deactivation stalling indefinitely, as the TM will wait for the responses from all delegates.

2.3 Bootstrap

Applications must supply a bootstrap sequence to the TM by implementing the following function:

```
#include "csr_tm_bluecore_bootstrap.h"

CsrUint8 **CsrTmBlueCoreGetBootstrap(CsrUint16 buildId, CsrUint16 *entries);
```

This function is called by the TM to obtain the bootstrap sequence. It must be implemented by the application. The function shall return an array of BCCMDs and the number of entries in the array. The array must be allocated by the application but will be owned by the TM. The BCCMDs will be sent in the order of placement in the array. Also note that this function must support being called multiple times returning the same output for each call.

Utility functions exist for building general SETREQ BCCMD messages as well as PSKEY SETREQ BCCMD messages.

```
CsrUint8 *CsrTmBlueCoreBuildBccmdSetMsg(CsrUint16 varId, CsrUint16 payloadLength,
const CsrUint16 *payload);
```

This function builds a SETREQ BCCMD message for the variable identified by `varId`. The BCCMD payload is provided in the `payload` parameter. The payload will be converted to XAP format by this function.

```
CsrUint8 *CsrTmBlueCoreBuildBccmdPsSetMsg(CsrUint16 key, CsrUint16 stores,
CsrUint16 psValuelength, const CsrUint16 *psValue);
```

This function builds a SETREQ BCCMD message with varId 0x7003, i.e. a persistent store command. The `key` parameter identifies the PSKEY of the element being accessed. The `stores` parameter controls the searching of the four component stores that make up the full Persistent Store. Stores field values are defined in csr_bccmd_prim.h.The PSKEY payload is provided in the `psValue` parameter. The payload will be converted to XAP format by this function.

3 TM Chip Manager API

An optional addition to the basic TM API enables monitoring of the chip/host interface. This additional API is called the TM Chip Manager API. The TM will detect if the chip becomes unresponsive – e.g. caused by failures in the chip/firmware that results in a chip panic. The TM will attempt to recover from such situations by trying to restart the host interface, assuming that the chip will restart after an unexpected reset. The TM API also allows registration of recovery handlers that will be signalled upon successful restart of the chip/host interface. A recovery handler exists e.g. for the Synergy Bluetooth technology. This handler will capture HCI commands sent during normal operation. Upon recovery, it will replay these commands to bring the chip in the same state as before the unexpected reset. Thus allowing the unexpected reset to appear as transparent as possibly to application code.

This feature cannot be enabled if CSR_BLUECORE_ONOFF is enabled, as the recovery from unsolicited chip reset is delegated to the controlling application, which will be able to deactivate and reactivate the BlueCore explicitly.

Applications can monitor the status of the chip/host interface by subscribing to status notifications. Notifications exist to report the following events:

- the chip/host interface has become unresponsive
- a restart of the chip/host interface has been successful
- a restart of the chip/host interface has failed
- a recovery handler has reported an error while attempting to re-establish state
- an initial read out of the chip panic and fault arguments

The TM Chip Manager API is enabled compile time by the compiler flag `CSR_CHIP_MANAGER_ENABLE`.

3.1 Chip Monitoring

The chip monitoring consists of pinging (periodically) the BlueCore® chip by sending it a specific BCCMD message. If the TM does not receive a reply on the ping message within a specified interval, it will assume that the chip has unexpectedly reset, and it will attempt to restart the host transport and rerun the bootstrap. The TM will notify any subscribers of the status on the chip/host interface. Finally, if the chip/host interface has successfully been restarted and the bootstrap has been rerun, then any registered recovery handlers will be started.

This section provides an overview of the chip monitoring scenarios and a description of the interface for retrieving status notifications.

Figure 4 shows the scenario of an unexpected chip reset detected by the ping mechanism. The figure shows that the application has initially subscribed for status notifications. In this scenario the application receives notifications when the reset is detected and when the transport has been restarted and the bootstrap and any recovery handlers have been run. In addition, the application receives a notification containing the read out of the chip panic and fault arguments. The read out is performed prior to the bootstrap being rerun.

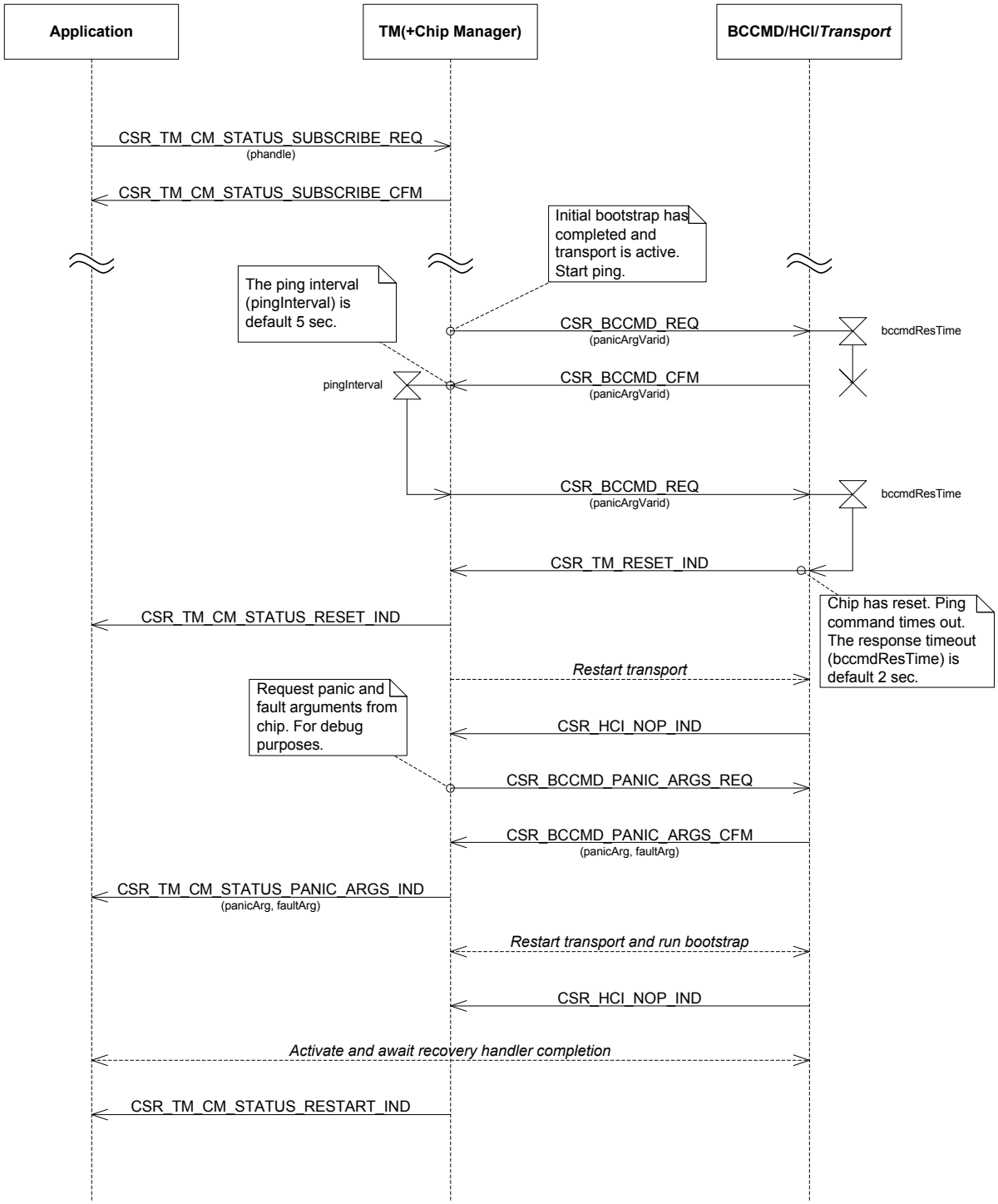


Figure 4: Chip Reset and Restart

In the description so far, the assumption has been that in the event of a reset, the chip actually restarts and is reachable from the host. In this situation, the TM Chip Manager will attempt to re-establish chip state and keep applications unaware of the reset. If however, the chip completely stalls, then the TM Chip Manager cannot attempt to do any recovery and it will simply report this event by a CSR_TM_CM_STATUS_PANIC_IND to the application. The application may then e.g. attempt to reset the product by any means specific to the particular platform. This scenario is shown in Figure 5.

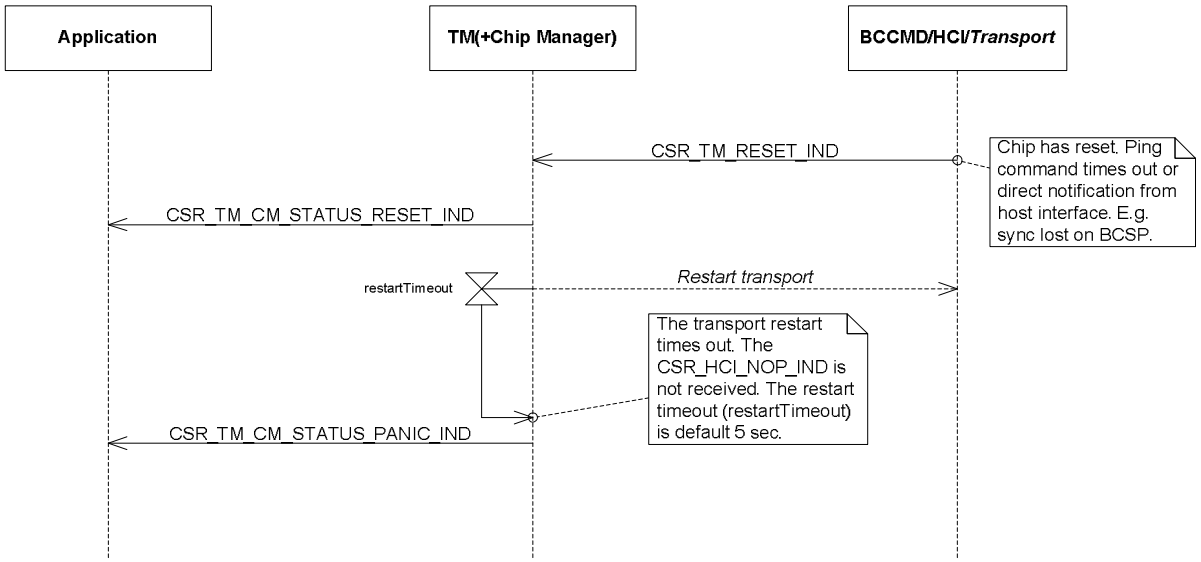


Figure 5: Failed Transport Restart

3.1.1 CSR_TM_BLUECORE_CM_STATUS_SUBSCRIBE

The TM Chip Manager API provides the following function for constructing and sending the status subscription primitive:

```
#include "csr_tm_bluecore_lib.h"
void CsrTmBlueCoreCmStatusSubscribeReqSend(CsrSchedQid phandle);
```

The parameters of the function are shown in Table 10.

Type	Parameter	Description
CsrSchedQid	Phandle	The identity of the calling task

Table 10: CsrTmBlueCoreCmStatusSubscribeReqSend

The parameters of the CSR_TM_BLUECORE_CM_STATUS_SUBSCRIBE_CFM primitive are shown in Table 11

Type	Parameter	Description
CsrTmBlueCorePrim	Type	Signal identity CSR_TM_BLUECORE_CM_STATUS_SUBSCRIBE _CFM
CsrResult	result	Always CSR_RESULT_SUCCESS

Table 11: CSR_TM_BLUECORE_CM_STATUS_SUBSCRIBE_CFM

3.1.2 CSR_TM_BLUECORE_CM_STATUS_UNSUBSCRIBE

Status subscriptions can be cancelled by the CSR_TM_BLUECORE_CM_STATUS_UNSUBSCRIBE_REQ. The following function must be used to construct and send the status unsubscribe primitive:

```
#include "csr_tm_bluecore_lib.h"
void CsrTmBlueCoreCmStatusUnsubscribeReqSend(CsrSchedQid phandle);
```

The parameters of the function are shown in Table 12.

Type	Parameter	Description
CsrSchedQid	Phandle	The identity of the calling task

Table 12: CsrTmBlueCoreCmStatusUnsubscribeReqSend

The parameters of the CSR_TM_BLUECORE_CM_STATUS_UNSUBSCRIBE_CFM primitive are shown in Table 13.

Type	Parameter	Description
CsrTmBlueCorePrim	type	Signal identity CSR_TM_BLUECORE_CM_STATUS_UNSUBSCRIBE _CFM
CsrResult	result	Always CSR_RESULT_SUCCESS

Table 13: CSR_TM_BLUECORE_CM_STATUS_UNSUBSCRIBE_CFM

3.1.3 CSR_TM_BLUECORE_CM_STATUS_(RESET,PANIC,RESTART)

The following TM Chip Manager status notifications are all simple indications with no parameters. The CSR_TM_BLUECORE_CM_STATUS_RESET_IND notifies that the TM has observed the chip as unresponsive. The CSR_TM_BLUECORE_CM_STATUS_PANIC_IND notifies that the TM could not restart the chip/host interface. It has not received a CSR_HCI_NOP_IND within a specified timeout. The CSR_TM_BLUECORE_CM_STATUS_RESTART_IND notifies that the chip/host interface has successfully been restarted and the bootstrap has been rerun, and the registered recovery handlers have been run.

The parameters of the above primitives are shown in Table 14.

Type	Parameter	Description
CsrTmBlueCorePrim	type	Signal identity – CSR_TM_BLUECORE_CM_STATUS_RESET_IND or CSR_TM_BLUECORE_CM_STATUS_PANIC_IND or CSR_TM_BLUECORE_CM_STATUS_RESTART_IND

Table 14: CSR_TM_BLUECORE_CM_STATUS_(RESET,PANIC,RESTART)_IND

3.1.4 CSR_TM_BLUECORE_CM_STATUS_PANIC_ARGS

Every time the chip/host interface is successfully restarted after an unexpected reset, the TM will read out the panic and fault arguments from the chip. The purpose of this is for debugging purposes only, as it may indicate the cause of the unexpected reset. The parameters of this indication are the panic and fault argument values read from the chip using the Panic Arg and Fault Arg BCCMDs, respectively.

Type	Parameter	Description
CsrTmBlueCorePrim	type	Signal identity CSR_TM_BLUECORE_CM_STATUS_PANIC_ARGS_IND
CsrResult	panicStatus	Status about the Panic Arg BCCMD
CsrUint16	panicArg	Panic Arg value retrieved
CsrResult	faultStatus	Status about the Fault Arg BCCMD
CsrUint16	faultArg	Fault Arg value retrieved

3.1.5 CSR_TM_BLUECORE_CM_PING_INTERVAL_SET

Applications can set the TM ping interval by the CSR_TM_BLUECORE_CM_PING_INTERVAL_SET_REQ. If the ping timer is running already it will be cancelled and restarted with the new value. If a zero value is provided, the ping timer will be cancelled and not restarted, effectively stopping the ping mechanism. It must be noted, that stopping the ping mechanism will prevent the TM to detect chip resets if the host transport/interface cannot provide a notification of this. For the transports supported by the TM only BCSP can provide this notification and only if the baudrate has not been changed during bootstrap. Figure 6 shows the sequence for setting the ping interval.

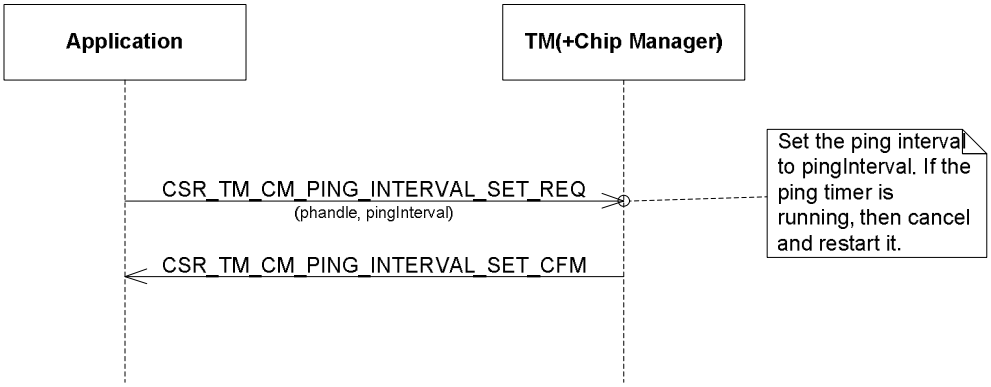


Figure 6: Setting Ping Interval

The following function must be used to construct and send the ping interval set primitive:

```
#include "csr_tm_bluecore_lib.h"
void CsrTmBlueCoreCmPingIntervalSetReqSend(CsrSchedQid phandle, CsrTime interval);
```

The parameters of the function are shown in Table 15.

Type	Parameter	Description
CsrSchedQid	Phandle	The identity of the calling task
CsrTime	Interval	The ping interval in usecs

Table 15: CsrTmBlueCoreCmPingIntervalSetReqSend

The parameters of the CSR_TM_BLUECORE_CM_PING_INTERVAL_SET_CFM primitive are shown in Table 16.

Type	Parameter	Description
CsrTmBlueCorePrim	type	Signal identity CSR_TM_BLUECORE_CM_PING_INTERVAL_SET _CFM
CsrResult	result	Always CSR_RESULT_SUCCESS

Table 16: CSR_TM_BLUECORE_CM_PING_INTERVAL_SET_CFM

3.2 Recovery Handlers

In the Synergy software there exist recovery handlers for the BCCMD task and for the Bluetooth HCI task. The BCCMD recovery handler will ensure that all BCCMD requests will receive a response, even in the case of chip resets. If the chip becomes unresponsive while there is an outstanding BCCMD request, the BCCMD recovery handler will provide the response for the outstanding BCCMD. It will always indicate that the BCCMD request has failed.

The Bluetooth HCI recovery handler resides within the Synergy Bluetooth component and it can only be activated if Synergy Bluetooth is part of the product. The HCI recovery handler will attempt to replay certain HCI commands (configuration commands) to the chip. In addition, it will send close indications for all active HCI ACL connections. This way, the recovery handler attempts to let the chip reset appear as transparent as possible to applications. Applications will only observe that the Bluetooth link has temporarily been broken.

The HCI recovery handler is enabled compile time by the compiler flag `CSR_CHIP_MANAGER_ENABLE` when compiling the Synergy Bluetooth component. This section provides an overview of the recovery handler scenarios and a description of the recovery handler interface.

3.2.1 CSR_TM_BLUECORE_CM_REPLAY_REGISTER

Figure 7 shows the Bluetooth HCI task and BCCMD task registering as recovery handlers. Note that in the interface naming, the term ‘replay’ is used instead of ‘recovery’. The primitive for registration is CSR_TM_BLUECORE_CM_REPLAY_REGISTER_REQ.

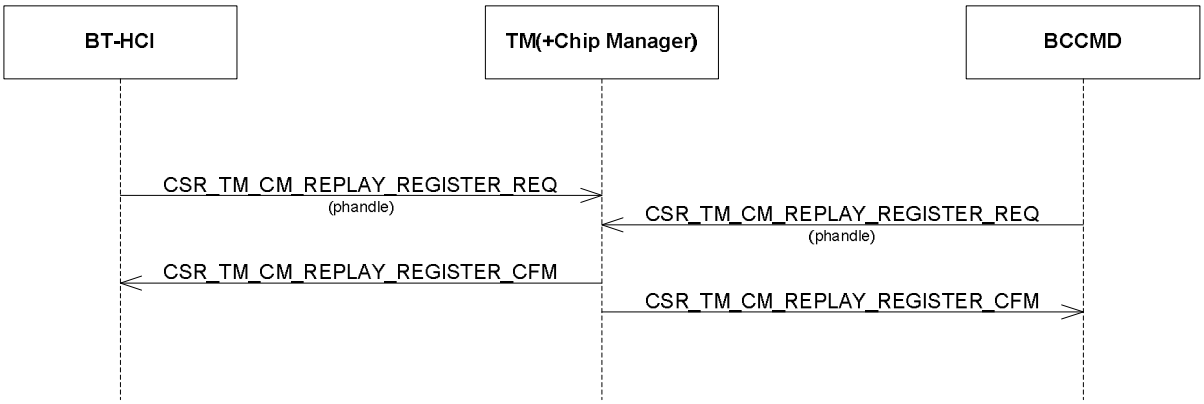


Figure 7: BT-HCI and BCCMD Replay Register

The following function is used to construct and send the replay registration primitive:

```
#include "csr_tm_bluecore_lib.h"

void CsrTmBlueCoreCmReplayRegisterReqSend(CsrSchedQid phandle);
```

The parameters of the function are shown in Table 17.

Type	Parameter	Description
CsrSchedQid	Phandle	The identity of the calling task

Table 17: CsrTmBlueCoreCmReplayRegisterReqSend

The parameters of the CSR_TM_BLUECORE_CM_ REPLAY_REGISTER_CFM primitive are shown in Table 18.

Type	Parameter	Description
CsrTmBlueCorePrim	Type	Signal identity CSR_TM_BLUECORE_CM_REPLAY_REGISTER_CFM
CsrResult	result	Always CSR_RESULT_SUCCESS

Table 18: CSR_TM_BLUECORE_CM_ REPLAY_REGISTER_CFM

3.2.2 CSR_TM_BLUECORE_CM_REPLAY_START

The HCI and BCCMD recovery handlers are started by the CSR_TM_BLUECORE_CM_REPLAY_START_IND primitive. This primitive is send when an unexpected chip reset has been detected by the TM and the bootstrap has been rerun and the host transport is activated.

The BCCMD recovery handler will simply provide failure responses to all BCCMD requests outstanding when the reset is detected. This way, it is guaranteed that all BCCMD requests will get a corresponding response, even in the case of chip resets.

The HCI recovery handler sends close indications for all active HCI ACL connections, and it replays HCI configuration commands to the chip.

Figure 8 displays the sequence for starting the HCI and BCCMD recovery handlers.

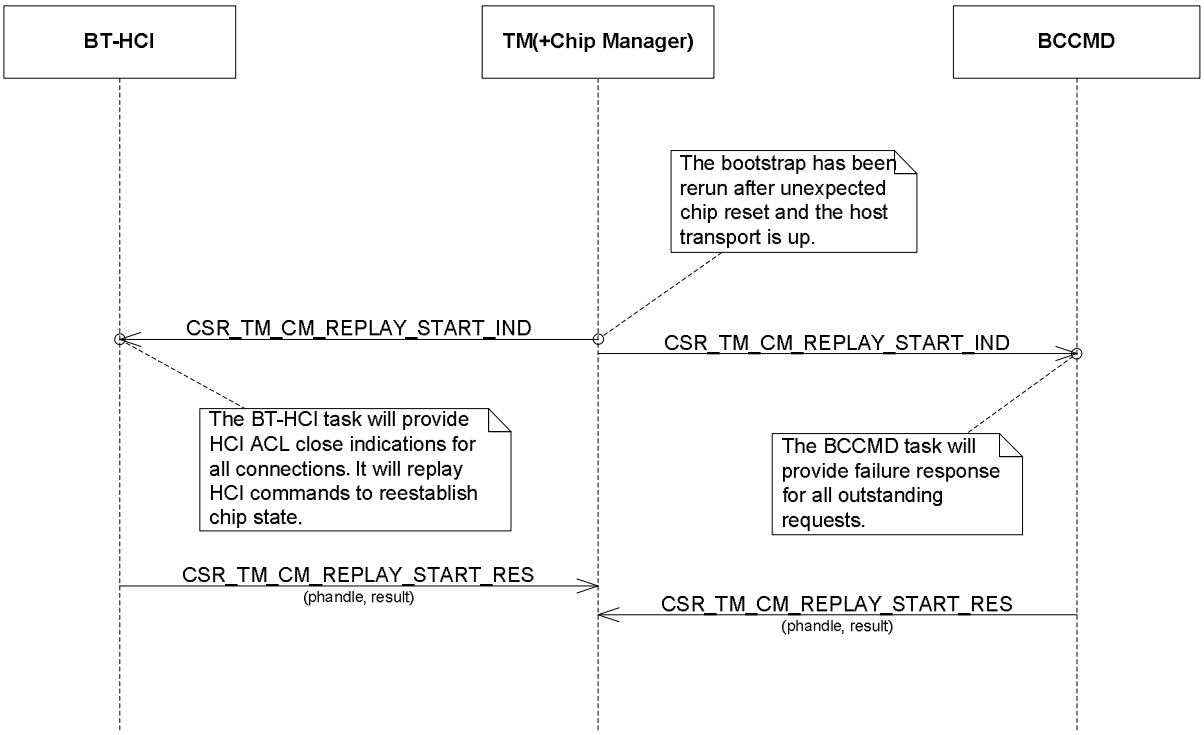


Figure 8: Starting Recovery Handlers

The parameters of the CSR_TM_BLUECORE_CM_REPLAY_START_IND primitive are shown in Table 19.

Type	Parameter	Description
CsrTmBlueCorePrim	Type	Signal identity CSR_TM_BLUECORE_CM_ REPLAY_START_IND

Table 19: CSR_TM_BLUECORE_CM_REPLAY_START_IND

The parameters of the CSR_TM_BLUECORE_CM_ REPLAY_START_RES primitive are shown in Table 20.

Type	Parameter	Description
CsrTmBlueCorePrim	type	Signal identity CSR_TM_BLUECORE_CM_ REPLAY_START_RES
CsrSchedQid	phandle	The identity of the responding task
CsrResult	result	Always CSR_RESULT_SUCCESS

Table 20: CSR_TM_BLUECORE_CM_PING_INTERVAL_SET_CFM

3.2.3 CSR_TM_BLUECORE_CM_STATUS_REPLAY_ERROR

During replay of HCI commands, the HCI recovery handler may receive a failure result to one of the replayed commands that had already succeeded prior to the chip reset. In this situation, the TM will simply notify status subscribers that the replay of this command failed, and the chip state could not be completely re-established after reset.

Figure 9 shows a scenario of a failing HCI command during replay. The application (status subscriber) obtains a notification by the CSR_TM_BLUECORE_CM_STATUS_REPLAY_ERROR_IND message, which contains the opcode of the failing command. The HCI recovery handler will continue to replay remaining commands even in the case that some fail. A CSR_TM_BLUECORE_CM_STATUS_REPLAY_ERROR_IND message will be send for each failing command. Also, notice that the replay will end with the HCI recovery handler sending CSR_TM_REPLAY_START_RES to the TM. The result code of this message will always by CSR_RESULT_SUCCESS, even if some commands have failed during replay.

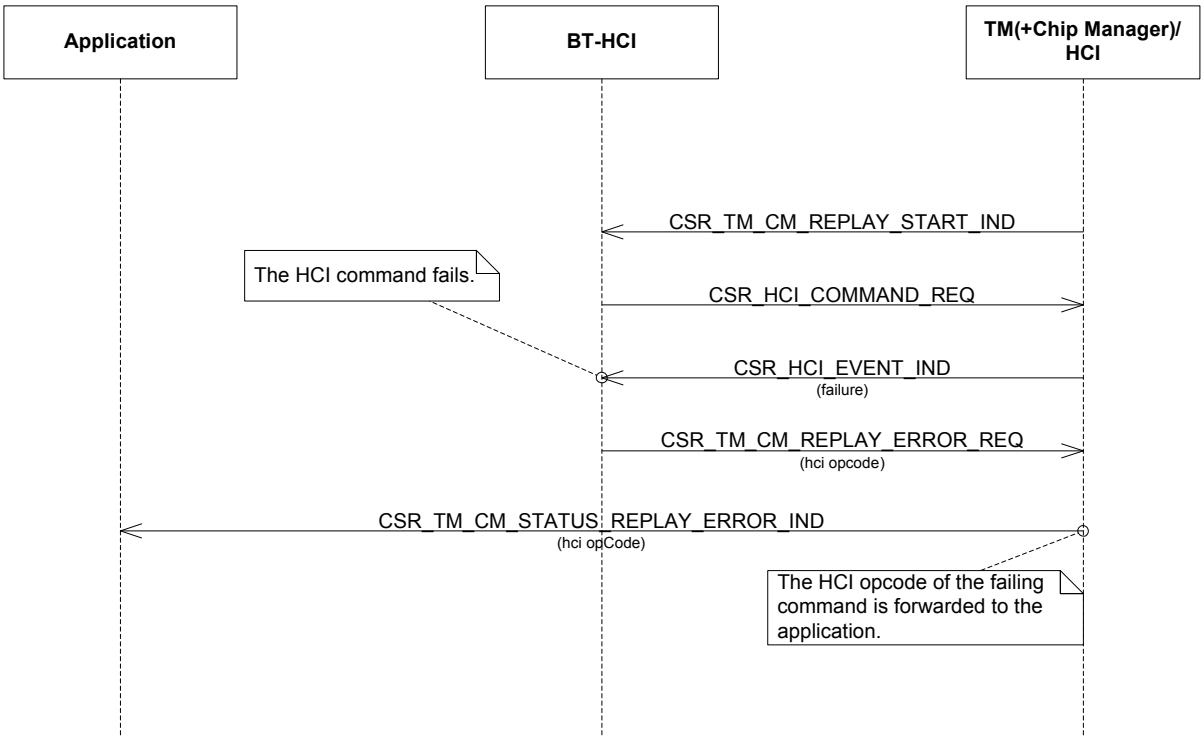


Figure 9: Failing Command in Replay

The parameters of the CSR_TM_BLUECORE_CM_STATUS_REPLAY_ERROR_IND primitive are shown in Table 21.

Type	Parameter	Description
CsrTmBlueCorePrim	type	Signal identity CSR_TM_BLUECORE_CM_STATUS_REPLAY_ERROR_IND
CsrUint16	hciOpcode	The HCI opcode of the failing command

Table 21: CSR_TM_BLUECORE_CM_STATUS_REPLAY_ERROR_IND

4 Document References

[SYN-FRW-SCHED-API]	*CSR Synergy Framework Scheduler API Description.* Doc. api-0004-sched.doc

Terms and Definitions

TM	Transport Manager
API	Application Programming Interface
BlueCore®	Group term for CSR's range of Bluetooth® wireless technology chips
CSR	Cambridge Silicon Radio

Document History

Revision	Date	History
1		First draft. Ready for review
2	07.02.2011	Updated with review comments

TradeMarks, Patents and Licences

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.