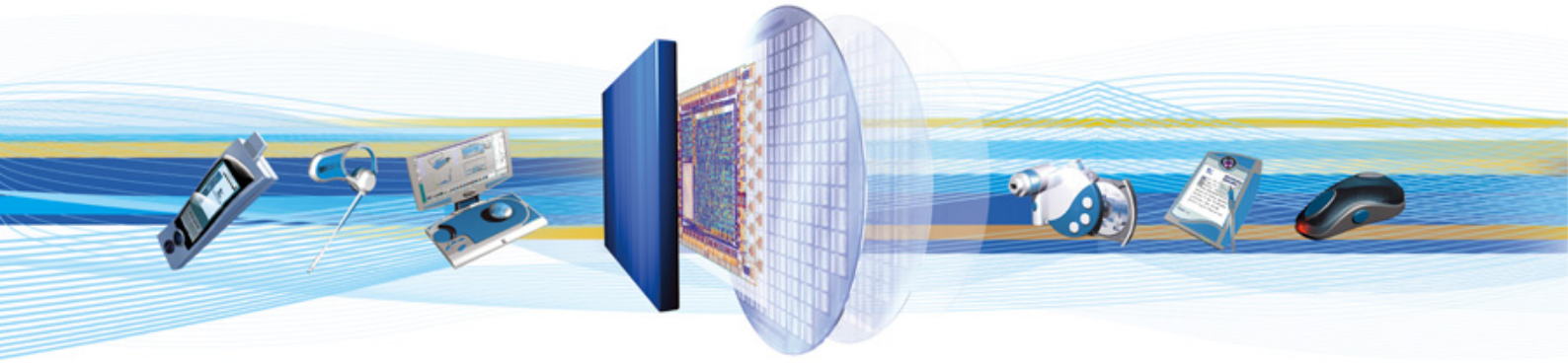
CSR Synergy Framework 3.1.1

Users Guide

November 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 Figures

List of Tables

1 Introduction

CSR Synergy Framework is a software layer providing a common infrastructure for easy integration of multiple wireless technologies. The Framework provides the core services of a virtual operating system, which shields technologies from variations in specific platforms. The Framework services include scheduling mechanisms, memory management, and transport protocols for communication with the CSR wireless chipsets. Services are also included for logging and various utilities like e.g. string handling and Unicode handling.

In Section 2, the architecture, components, and directory structure of the CSR Synergy Framework are presented. Section 2.4 describes the process of building and configuring the Framework.

2 Framework Architecture

The CSR Synergy Framework software consists of a set of components with well defined programming APIs. Some of the APIs have generic implementations provided by the Framework. These form the *Generic Support Package (GSP)* of the Framework. Other APIs are inherently platform dependent and require a platform specific implementation, also called a *port*. Such a port forms a *Board Support Package (BSP)* for the particular platform. Figure 1 shows the Framework architecture with GSP and BSP components in light blue and dark blue colours, respectively.

The BSP components consist of the CSR OS abstraction APIs, the log transport API, the utilities API, and separate APIs for all supported transport drivers. The OS abstraction APIs consists of a core scheduler API for task scheduling, message forwarding, timer handling, memory management, and background interrupts. Also it consists of APIs for retrieval of system time, for panic handling, and for exception handling. The BSP APIs must be ported to the real host platform under consideration. The CSR Synergy Framework release provides the BSP APIs as well as reference implementations for PC Windows NT, PC Linux, and BDB2 Nucleus. The BDB2 platform is a CSR internal development platform containing an ARM926EJ-S core and a large number of peripherals.

The GSP components all consist of both an API and a generic implementation. The components include transport protocols BCSP, H4DS, USB, and TYPE-A, host controller interface, command interface BCCMD, VM, HQ, and FastPipe, and logging. All Framework components are provided as ANSI-C source and header files. In the following sections, an overview of the various components is provided.

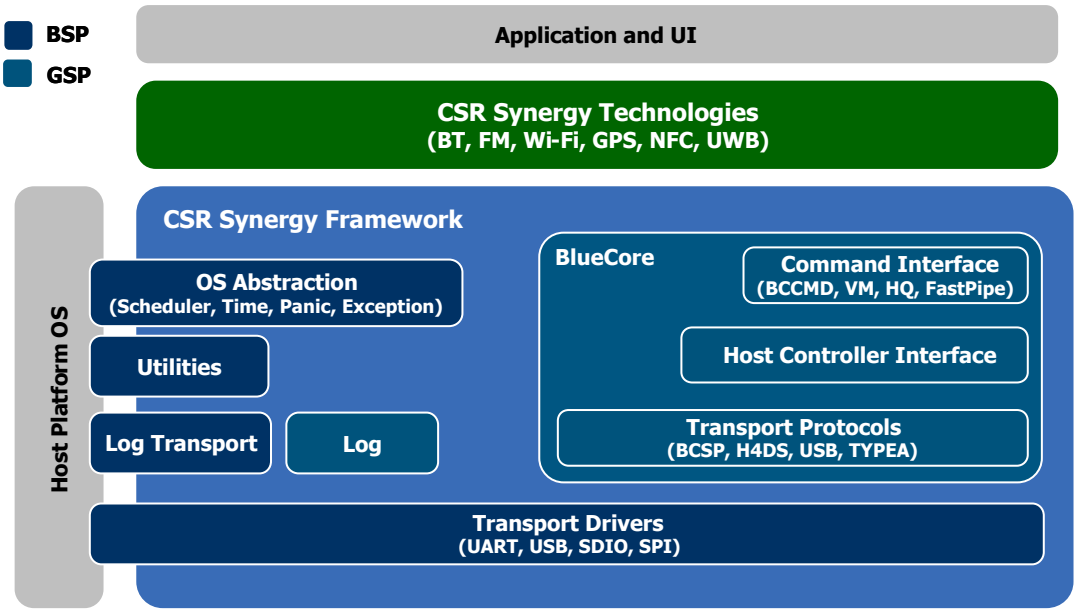


Figure 1: CSR Synergy Framework Architecture

2.1 BSP Components

This section presents an overview of the Framework BSP components.

2.1.1 Scheduler API

The Scheduler API consists of the set of the services listed below. The API must be ported to the platform/OS used.

- Task scheduling
- Message forwarding

- Timer handling
- Background interrupts

In the following, the basic functionalities of the Scheduler API are presented in overview.

Note: For detailed description of the Scheduler API, its porting requirements, and its Framework reference implementations, please refer to *CSR Synergy Framework Scheduler API* [SYN-FRW-SCHED-API].

Task Scheduling

The system *scheduler* is at the very core of all software developed on top of the CSR Synergy Framework. It organises code as a set of *tasks* passing *messages* to each other. Each task is attached to one *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.

Message Forwarding

As mentioned above, a scheduler task is partly identified by its message queue and message handler function. The message queue accumulates messages, and the message handler consumes messages from its queue and acts upon them. Tasks should normally obey the convention that when a message is allocated, 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 free the message storage.

The message forwarding API consists of functions for:

- putting messages in a queue
- getting messages from a queue
- retrieving task queue id

The above API is considered for communication of messages between tasks inside the same scheduler context. Thus, it may be seen as an *internal* messaging API. The code structure for a given implementation may consist of parts running both inside and outside scheduler context, or of parts running in separate scheduler processes. To separate the communication of external and internal messages, an *external messaging* API exists. This API is e.g. 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.

Timer Handling

The timer API consists of functions for:

- generating timed events (calling functions after a given time)

Background Interrupts

The scheduler contains a mechanism for tasks to signal the execution of certain processing at the next available time when no tasks are running. This may for example be used by a device driver to signal that data is ready without requiring the scheduler to retrieve the data right away.

The background interrupt API consists of functions for:

- registering background interrupt handler functions with the scheduler

- unregistering background interrupt handler functions
- signalling background interrupt handlers to run

Scheduler API Headers

The Scheduler API is contained in the following Framework header files

csr_sched.h	Task scheduling, Message forwarding (internal), Timer handling, Background interrupts
csr_msg_transport.h	Message forwarding (external)

2.1.2 Memory Management (Pmem) API

The Memory API Pmem consists of functions for:

- allocating and freeing memory

Note: For detailed description of the Memory Management API, its porting requirements, and its Framework reference implementations, please refer to *CSR Synergy Framework Pmem API* [SYN-FRW-PMEM-API].

Pmem API Headers

The Pmem API is contained in the following Framework header file.

csr_pmem.h	Memory management

2.1.3 Time API

The Time API consists of functions for:

- Retrieval of system time
- Retrieval of UTC time

Note: For detailed description of the Time API, its porting requirements, and its Framework reference implementations, please refer to *CSR Synergy Framework Time API* [SYN-FRW-TIME-API].

Time API Headers

The Time API is contained in the following Framework header file.

csr_time.h	System and UTC time

2.1.4 Panic API

The panic handling API is used for signalling and handling events unrecoverable by the scheduler or other Framework components. E.g. memory exhaustion or chip communication failure. The appropriate action to take in these situations, e.g. restart of the system, may depend on factors unknown to the Framework. This could be e.g. which technologies are in use, or some details of the application.

The panic handling API consists of a function for:

- generating panic events

Note: For detailed description of the Panic API, its porting requirements, and its Framework reference implementations, please refer to *CSR Synergy Framework Panic API* [SYN-FRW-PANIC-API].

Panic API Headers

The Panic API is contained in the following Framework header file.

csr_panic.h	Panic handling

2.1.5 Exception API

The exception handling API is to be used only for debugging purposes. The CSR Synergy Framework code can be configured as to whether is will use the exception handling component or not. The exception handling is by default turned off. The *panic API* is used for reacting upon e.g. memory exhaustion or chip communication failure in production code.

The exception handling API consists of functions for:

- registering exception handler functions
- raising exceptions

Note: For detailed description of the Exception API, its porting requirements, and its Framework reference implementations, please refer to *CSR Synergy Framework Exception API* [SYN-FRW-EXCEPTION-API].

Exception API Headers

The Exception API is contained in the following Framework header file.

csr_exceptionhandler.h	Exception handling

2.1.6 Utilities API

The Utilities API must be ported to the platform/OS used. It is part of the Framework BSP interface. The utility API consists of a small number of functions for:

- bit operations, endian and base conversions, and string operations

Utilities API Header

csr_util.h	Utilities

2.1.7 Transport Drivers API

The transport drivers API provide the driver interface for the low-level communication media between the host software and the CSR wireless chip firmware. The API for the particular transport driver(s) supported and used by the platform must be ported. The API is part of the Framework BSP interface. The Framework includes transport driver APIs for:

- UART, USB, SDIO, SPI

The transport drivers can be seen as the low-level and platform-dependent parts of specific *host transport protocols*. The transport driver APIs are used by the transport protocols. The host transport protocols using the transport drivers are:

- BCSP and H4DS using UART
- USB (driver interface, no additional protocol)
- Type A using SDIO/SPI

The CSR Synergy Framework provides generic implementations (GSP) of the transport protocols BCSP, H4DS, and Type A. An overview of these protocols is provided in Section 2.2.1.

Note: For detailed description of the Transport Drivers API, its porting requirements, and its Framework reference implementations, please refer to *CSR Synergy Framework UART API* [SYN-FRW-UART-API] and *CSR Synergy Framework SDIO API* [SYN-FRW-SDIO-API].

UART

When the host transport communication is based on UART one of two communication protocols are used on the interface, either BCSP or H4DS. BCSP defines logical channels, which can be used for routing of signals between the host software and the CSR chip. H4DS is a simple protocol that assumes error free communication and thus, has no retransmission capabilities.

The UART driver API consists of functions for:

- starting the transport driver
- stopping the transport driver
- sending data
- receiving data

USB

When using USB as the host transport, there is no additional protocol on top of the driver interface. Porting the USB interface is similar to porting the UART interface. The API consists of functions for:

- starting the transport driver
- stopping the transport driver
- sending data
- receiving data

SDIO/SPI

When using SDIO or CSPI as the host transport, the communication protocol Type A is used on the interface. Regardless of whether SDIO or CSPI is used as the host transport, the CSR Synergy Framework provides *CSR SDIO* as host interface API. CSR SDIO manages one or more SDIO and/or CSPI devices.

The CSR SDIO API consists of functions for:

- registering *function drivers* supporting one or more SDIO/CSPI functions with the SDIO Manager.
- byte oriented data transfer (read/write)
- bulk data transfer (read/write)

Transport Drivers API Headers

The Transport Drivers API is contained in the following Framework header files

csr_serial_com.h	UART

csr_usb_com.h	USB
csr_sdio.h	CSR SDIO

2.1.8 Log Transport API

The Framework log component consists of both a generic, GSP, part and a platform dependent part, BSP, to be ported. The generic log interface provides a simple framework for logging various types of events. It will format log messages and pass them on to any registered *log transports*. It allows multiple log transports to be attached. The log transport is responsible for e.g. transferring the log data to a file or sending it over a serial cable or similar. The log transport interface is platform dependent, BSP, and must be ported.

The generic log API provides functionality for:

- logging all HCI communication and all interaction in the CSR Synergy Framework and protocol stack

The log transport API provides functionality for:

- writing log data to the transport
- file related operations like opening, closing, and flushing – only relevant for platforms with file system support

Note: For detailed description of the Log Transport API, please refer to *CSR Synergy Framework Log Transport API* [SYN-FRW-LOGTRANS-API].

Log Transport API Headers

csr_log_transport.h	Log transport interface

2.2 GSP Components

This section presents an overview of the Framework GSP components.

2.2.1 Transport Protocols API

The CSR Synergy Framework provides generic implementations, GSP, of the transport protocols BCSP, H4DS, and Type A.

BCSP

The BlueCore® Serial Protocol, BCSP, adds possibilities for *logical communication channels.* The logical communication channels are used for routing signals to and from the host and host controller, and they make it possible to establish a communication path between components in the host and host controller. BCSP makes it possible to use the low-power modes of the BlueCore® chip. Furthermore, BCSP offers reliable communication with retransmission of messages in case of corruption. The BCSP protocol is a CSP specific protocol resembling the H5 / 3-Wire UART protocol.

H4DS (H4 Deep-Sleep)

One of the HCI protocols defined in the Bluetooth 2.1 specification and later, is the HCI UART Transport Layer. This protocol is defined in Part H4 in the Bluetooth specification and is commonly known simply as H4. The H4 implementation in CSR Synergy Framework uses a proprietary Deep Sleep extension that makes it possible to use the low-power modes of the BlueCore® chip. H4DS is an alternative to BCSP for handling the communication between the host and the host controller. One of the main differences between H4DS and BCSP is that BCSP facilitates retransmission of corrupt messages whereas H4DS assumes error free communication,

meaning that H4DS might have a slightly higher throughput, but will fail if a transmission error happens. Due to this BCSP is the preferred communication over serial lines.

Type A

The SDIO Bluetooth Type-A protocol is used for running the HCI protocols over a SDIO Card interface. The SDIO Bluetooth Type-A protocol that is implemented in CSR Synergy Framework is according to the standard specified in “SDIO Card Part E2, Type-A Specification for Bluetooth version 1.00 September 2002” [SPEC-TYPE-A]. The bottom of the SDIO Bluetooth Type-A protocol implementation interfaces to the SDIO Manager interface (see Section 2.1.7) which makes it easy to port the SDIO Bluetooth Type-A onto a new platform. The Type-A protocol can be transported on either SDIO or CSPI (CSR-SPI).

Transport Manager

The Framework Transport Manager (TM) task is responsible for controlling the transports. The TM task provides functionality for:

- initializing transport protocols
- activating(starting) transport protocols
- running bootstrap sequence for BlueCore®

Transport Manager API Headers

csr_tm_bluecore_lib.h	TM message send lib
csr_tm_bluecore_prim.h	TM message primitives
csr_tm_bluecore_bcsp.h	Initialization of BCSP
csr_tm_bluecore_h4ds.h	Initialization of H4DS
csr_tm_bluecore_type_a.h	Initialization of TYPE A

2.2.2 Host Controller Interface API

The host controller interface API provides functionality for sending and receiving data to and from a BlueCore® chip on a set of predefined transport channels. The host interface towards the BlueCore® can be of either HCI of RFC types. In an HCI interface type for Bluetooth, the upper core stack is located on the host platform and HCI provides the interface to the lower core stack layers. In the RFC interface type, all core stack layers in the BlueCore®. The CSR Synergy Framework provides generic implementations, GSP, of the host interface.

The host interface API provides functions for:

- sending messages on specific transport channels
- registering message handlers to handle messages received on specific transport channels

Host Interface API Headers

csr_hci_{lib, prim}.h	Host Interface interface
csr_sco_audio.h	Handling SCO data

2.2.3 Command Interface API

The command interface API provides functionality for controlling and monitoring a BlueCore® chip. The interface consists of four APIs: BCCMD, HQ, VM, and FastPipe. The CSR Synergy Framework provides generic (GSP) implementations of all three.

Note: For detailed description of the Command Interface API, please refer to *CSR Synergy Framework BCCMD API* [SYN-FRW-BCCMD-API], *CSR Synergy Framework HQ API* [SYN-FRW-HQ-API], *CSR Synergy Framework VM API* [SYN-FRW-VM-API], and *CSR Synergy Framework FastPipe API* [SYN-FRW-FP-API].

BCCMD

The BCCMD (BlueCore® Command) protocol interfaces to a command interpreter on the BlueCore® chip. The command interpreter presents commands that allow monitoring and controlling of the chip. The command set is not part of the Bluetooth® standard.

When using BlueCore® ROM chips a *bootstrap procedure* has to be performed. The bootstrap is a sequence of BC commands that set the required parameters in the chips persistent storage when the chip powers up.

The BCCMD API provides functionality for:

- transferring various types of BCCMD messages

HQ

The BlueCore® chip can signal, control, and monitor its Bluetooth host through the command interpreter protocol, Host Query (HQ). The client (the BlueCore® chip) sends requests to the server (the host), and the server services the requests and returns responses to the client. The server presents its resources as a simple name/value database. A name is referred to as a *varId*. The protocol is almost entirely used by the chip to indicate certain values to the host. The hosts response it typically a “got it” message back to the chip. Host applications using the HQ API can register themselves to receives indications from the HQ component upon chip requests/indications.

The HQ API provides for:

- Registering an arbitrary number of application handles, each subscribing to an arbitrary number of varIds
- Error handling of varIds FAULT and DELAYED_PANIC unless these varIds are registered by an application

VM

The VM (Virtual Machine) interface provides functionality and message interface used for receiving and sending Virtual Machine (VM) commands from and to the Virtual Machine within the BlueCore® chip. The Virtual Machine embedded on the BlueCore® is capable of running special made applications like the AV-router, echo cancelation etc. The VM protocol is not part of the Bluetooth standard.

FastPipe

The FastPipe interface provides functionality and message interface for fast data transfer to and from a BlueCore® chip via UART interface. FastPipe allows the host to send data to BlueCore® using fast, flow controlled, channels that coexist with the HCI channels. The data is sent over ACL channels that FastPipe owns. The BlueCore® has hardware acceleration for data RX and TX on ACL channels, which is why the data is sent this way. The FastPipe protocol is not part of the Bluetooth standard.

Command Interface API Headers

csr_bccmd_{lib, prim}.h	BCCMD interface
csr_hq_{lib, prim}.h	HQ interface

csr_vm_{lib, prim}.h	VM interface
csr_fp_{lib, prim}.h	FastPipe interface

2.2.4 Log API

As described in Section 2.1.8 the log component consists of both a generic, GSP, part and a platform dependent part, BSP, to be ported. The generic log interface provides a simple framework for logging various types of events.

The generic log API provides functionality for:

- logging all HCI communication and all interaction in the CSR Synergy Framework and protocol stack

Log API Header

csr_log.h	Log interface

2.3 Directory Structure

This section describes the CSR Synergy Framework source code directory structure. The directory structure has been made to reflect the Framework architecture the best way possible. The following criteria form the fundamental idea behind the structure:

- Clear separation of generic and platform specific APIs. This in order to make it as simple as possible for users to identify the Framework parts which require porting.

The root directory structure is described in detail in Table 1. The directory structure for the bdb2 reference port directory structure is shown in Table 2. All reference ports follow the same structure.

/**bsp**	Interfaces and reference implementations for APIs with porting requirements
/bsp/**inc**	Platform independent BSP interface specification. It contains the fixed function prototypes, typedefs, and macros for all BSP APIs. These files specify the fixed interface and are not subject to change.
/bsp/**ports**	BSP reference ports containing implementation of the APIs in /bsp/inc
/bsp/ports/**bdb2**	Reference port for bdb2 platform
/bsp/ports/**pclin**	Reference port for PC Linux platform
/bsp/ports/**pcwin**	Reference port for PC Windows platform
/bsp/spec/**test**	Test cases for conformance testing BSP API implementation
/**config**	Configuration files for CSR Synergy Framework builds
/**doc**	Users guide and release note documentation
/doc/**bsp**	Interface specifications for BSP interfaces
/doc/**gsp**	Interface specifications for GSP interfaces
/gsp/**inc**	The GSP interface specification.

/gsp/**src**	GSP implementation
/gsp/src/**inc**	Private GSP header files
/**scripts**	Scripts (generic) used by the CSR Synergy Framework build system
/**output**	Libraries for GSP components
/**tools**	Tools used by the CSR Synergy Framework build system

Table 1: CSR Synergy Framework Root Directory Structure

/bsp/ports/bdb2/**inc**	Platform specific types. E.g. basic types like integer types that are platform specific.
/bsp/ports/bdb2/inc/**platform**	Platform dependent BSP interfaces. It must only be used from platform dependent code – e.g. specific application code. It must not be used from generic code e.g. Synergy Technologies or generic CSR Synergy Framework components.
/bsp/ports/bdb2/**output**	Libraries for BSP reference implementation
/bsp/ports/bdb2/**scripts**	Scripts (platform specific) used by the CSR Synergy Framework build system
/bsp/ports/bdb2/**src**	Code for the BSP reference implementation

Table 2: CSR Synergy Framework BSP Directory Structure

2.4 Porting Steps

As described before, all the Framework interfaces that are designed to have platform specific implementations are located in the BSP interfaces. This set of interfaces is kept to a minimum and is clearly distinguished in the Framework directory structure, as explained in Section 2.3.

Testing Scheduler API Port

The Scheduler API is at the very core of the CSR Synergy Framework. Therefore, the Framework includes a set of test cases to verify conformance of a given Scheduler API implementation to the interface specified. Running and passing these tests is a requirement when implementing the Scheduler API. The tests are located in the folder:

- /bsp/spec/test/coal

The test suite consists of the following test cases for:

- Message forwarding
- Timer handling
- Background interrupts

For a detailed description, please refer to the bsp/spec/test/readme.txt. The test cases can be build and run for all Framework reference ports using the following command in the bsp/spec/test folder.

```
make all TARGET=<target-name>
```

Where <target-name> may be one of pcwin-nt-x86, pclin-2.6-x86, or bdb2-nucleus-arm5tej. Targets are further explained in Section 3.1.

2.5 Determining Dependencies

The CSR Synergy Framework contains a tool known as depmap, which makes it possible to reveal the (link-time) dependencies on, of and between individual components. The tool is a Perl script, which is located at tools/depmap/depmap.pl. Using depmap allows answering questions about which components are required to port a specific generic component, or how generic components depend on each other. It also reveals the reverse dependencies, for example listing the components that depend on a given component. The tool analyses object files contained in libraries, under the assertion that a single object library corresponds to one component. The basic unit is the individual symbols in the global namespace that are either defined or referenced in the individual object files. It is possible to specify an option that collapses the symbols into the libraries that define the symbol, which makes it easier to get an overview of the dependencies. The full view, however, is useful to determine exactly which symbols causes the dependency on a component/library.

Before using depmap it is necessary to build the entire code base to generate the object libraries which are the subject of the analysis. Please refer to section **Error! Reference source not found.**3 for instructions for performing the build. It is also required to have binutils installed as depmap uses the nm utility to analyse the object libraries.

The tool can either be used directly or through the build system. The latter approach makes it somewhat easier because the build system automatically provides some of the information, so that none or only a few additional command line arguments are necessary. In the following, both approaches will be described.

For complete information about using the Perl script directly, please refer to the tools/depmap/depmap.pl file. It contains a comment header which describes all the options. The only required option is the --libpaths option. It is used for specifying a comma-separated list of paths to directories that contain object libraries to analyse. The specified directories must contain nothing but object libraries. In the absence of any other options, depmap will proceed to analyse all the object libraries in the specified directories and then emit a complete list of all dependencies on, of and between all the object libraries. Further options can be specified to limit the output to only the information desired. For example the --maplibs option can be used to limit the output to only contain dependencies relating to the specified libraries (known as the maplibs). The --maplibs option accepts a comma-separated list of any mix of library names (with or without path) and directories containing libraries (and nothing but libraries). The following example will search all object libraries in the directories output/default/pcwin-nt-x86/lib and bsp/ports/pcwin/output/default/pcwin-nt-x86/lib and report all dependencies relating to the object library csr_arg_search.lib:

```
framework> tools/depmap/depmap.pl --libpaths output/default/pcwin-nt-
x86/lib,bsp/ports/pcwin/output/default/pcwin-nt-x86/lib --maplibs
csr_arg_search.lib
        csr_app_main.lib -> CsrArgSearch (csr_arg_search.lib)
        csr_app_main_bluecore.lib -> CsrArgSearch (csr_arg_search.lib)
        csr_arg_search.lib -> CsrStrCmp (csr_util.lib)
        csr_arg_search.lib -> CsrStrLen (csr_util.lib)
        csr_arg_search.lib -> CsrStrNCmp (csr_util.lib)
        csr_main.lib -> CsrArgSearchInit (csr_arg_search.lib)
```

This output shows that the csr_arg_search.lib library itself depends on three symbols that are contained in the csr_util.lib library. Consequently, it is necessary to port this component to satisfy the dependencies of the csr_arg_search library. The output also shows that three other components depend on csr_arg_search.lib.

As it is pretty tedious to enter the arguments for the --libpaths and --maplibs parameters, the tool has been integrated into the build system in the form of a globally present makefile target named depmap, which will automatically provide default arguments for these parameters. This makes it much easier to invoke the tool and quickly retrieve the required information. The use of this makefile target is described in the global_rules.mk file in the root of the Framework. It contains a comment header that describes the options as well as some basic examples. The most useful way to use this is to invoke it from any makefile in a directory containing a component to investigate dependencies for:

```
framework/gsp/src/utils/arg_search> make depmap
        csr_app_main.lib -> CsrArgSearch (csr_arg_search.lib)
        csr_app_main_bluecore.lib -> CsrArgSearch (csr_arg_search.lib)
        csr_arg_search.lib -> CsrStrCmp (csr_util.lib)
        csr_arg_search.lib -> CsrStrLen (csr_util.lib)
        csr_arg_search.lib -> CsrStrNCmp (csr_util.lib)
        csr_main.lib -> CsrArgSearchInit (csr_arg_search.lib)
```

The depmap makefile target will automatically determine the argument to use for both --libpaths and --maplibs. The latter will be assigned to the value of LIB_NAME in the corresponding makefile. This means that this makefile target will automatically determine which library corresponds to the component of the current directory. As a special case, the top level makefile which does not itself correspond to any library contain a definition of LIB_NAME that causes dependencies for all GSP components to be emitted when the depmap makefile target is executed at the root of the Framework.

The argument passed to --libpaths is contained in the DEPMAP_LIBPATHS makefile variable, and the argument to --maplibs is contained in the DEPMAP_MAPLIBS variable. Both of these can be overridden if desired. In addition, extra arguments can be passed to depmap by overriding the DEPMAP_ARGS variable, which is empty by default. Please refer to tools/depmap/depmap.pl for information about which extra arguments are possible. An example is the use of the --collapse option and the --noreverse options, which causes the symbols to be collapsed into libraries and the reverse dependencies to be hidden:

```
framework> make depmap DEPMAP_ARGS="--collapse --noreverse"
        csr_arg_search.lib -> csr_util.lib
```

While the textual information generated by depmap is useful to determine the dependencies, it can sometimes be an advantage to visualize the relationships graphically. For that purpose, the depmap tool is capable of emitting dot format output suitable for use by Graphviz by using the --dot option. See http://www.graphviz.org/ for procuring and using Graphviz. Assuming Graphviz is installed, it is possible to generate a PDF file showing all dependencies on, of and between all GSP components by executing the following command in the root of the Framework:

```
framework> make depmap DEPMAP_ARGS="--collapse --dot" | neato -Tpdf >
depmap.pdf
```

Note that neato is a command which is part of the Graphviz installation. The result will be written to the depmap.pdf file which can be opened in any PDF reader. By tailoring the DEPMAP_ARGS, DEPMAP_MAPLIBS and DEPMAP_LIBPATHS variables it is possible to customise the generated visualisation to any desired view. For example, to restrict the view to a certain library or set of libraries, override the DEPMAP_MAPLIBS as described above.

3 Build System

This section describes the CSR Synergy Framework build system used for building the CSR Synergy Framework GSP libraries and the BSP reference implementation libraries. The build system is based on **Unix style build platforms (Linux or Cygwin)** with the **GNU make** utility.

3.1 Targets

The Framework build system supports building the Framework code for different *targets*. A target is a specific combination of a *target platform*, a *target OS*, and a *target architecture*. An example target supported by the Framework build system is:

- **pclin-2.6-x86:** Linux based PC platform with 2.6 OS and x86 architecture

Table 3 shows the list of targets, build platforms and compiler versions supported by the Framework build system. For the bdb2 target platform, the supported cross-compilation platform is a Windows based PC platform with Cygwin using the RVCT/ADS compilers.

Target (platform-os-arch)	Build Platform	Compiler Version
bdb2-nucleus-arm5tej	pcwin-nt-x86 + Cygwin	RVCT/ADS v. 1.2, 2.2, 3.0, 3.1
pclin-2.6-x86	pclin-2.6-x86	gcc v. 3.3.3 -> 4.0.0
pcwin-nt-x86	pcwin-nt-x86 + Cygwin	MS Visual C/C++ 6.0 (12.00.8804)

Table 3: Supported targets, build platforms, and compiler versions

The build system exposes a symbol TARGET to be used in build commands to choose the target. Like:

```
make all TARGET=bdb2-nucleus-arm5tej
```

The build system internally breaks the TARGET symbol into symbols TARGET_PLATFORM, TARGET_OS, and TARGET_ARCH such that:

- TARGET = <TARGET_PLATFORM>-<TARGET_OS>-<TARGET_ARCH>

The target symbols are used by the build system in two ways. To determine location of target specific sources and target specific build system configuration. In addition, to perform conditional compilation of target sources.

The target specific sources are located by the TARGET_PLATFORM symbol. This symbol identifies the root folder of the BSP interface supporting this platform. Specifying e.g. TARGET=bdb2-nucleus-arm5tej in the build command will imply TARGET_PLATFORM=bdb2. This again implies that the build system will look for sources and target specific configuration in the BSP folder defined as:

- BSP_ROOT := $(FW_ROOT)/bsp/ports/bdb2

Where FW_ROOT is locating the root of the CSR Synergy Framework installation. The FW_ROOT is set automatically when building the Framework and need not be specified by users.

The target specific build system configuration file is located as:

- $(BSP_ROOT)/scripts/target-$(TARGET).mk

This configuration contains parameters like default compiler, default target libraries, and default target CFLAGS. These parameters are supposed to be fixed and chosen to be optimal for the given target platform. Other parameters are intended to change more frequently in e.g. customization of development builds where enabling/disabling features like debug symbols, log generation, etc. may change. Section 3.3 describes in more detail how build configuration is performed.

3.1.1 Known Compiler Issues

The following compilers are known to generate wrong code in the specified configuration:

- ARM RVCT3.0 build 688: When using -O3 -Otime the compiler may generate misbehaving code for certain valid loop constructs. The recommended workaround is to use -O2 -Otime, for this particular compiler.

3.2 External Tools and Environment

This section describes prerequisites on external tools or environment settings needed in order to build CSR Synergy Framework.

Note: Most prerequisites are only relevant for CSR internal users developing new Synergy components. Such users will work on a CSR Synergy Framework developers release containing tools and scripts to e.g. auto-generate certain files. Most prerequisites relates to these scripts and tools. Please note, that for non-developers (customer) releases all the required auto-generated files are pre-generated and part of the release.

In the following, each prerequisite is marked with the release type and build platform to which it is relevant.

3.2.1 Perl

Build Platform	**pcwin-nt-x86 + Cygwin, pclin-2.x-x86**
Release Type	**developer**
Description	Cygwin and Linux build environments require **Perl v. 5.8 or later**.

3.2.2 Yacc

Build Platform	**pcwin-nt-x86 + Cygwin**
Release Type	**developer**

Description	The Cygwin build environment requires the **yacc** tool to be part of the Cygwin installation. **yacc v. 2.3 and v. 2.4** is known to work – other versions are untested. Check the version by:
	
	`$ yacc -V`
	
	`bison (GNU Bison) 2.3`

3.2.3 Shellopts

Build Platform	**pcwin-nt-x86 + Cygwin**
Release Type	**developer**

Description	The Cygwin build environment requires `igncr` to be part of the `SHELLOPTS` environment variable. Check by:
	
	`$ echo $SHELLOPTS`
	
	`braceexpand:emacs:hashall:histexpand:history:`**`igncr`**`:interactive-comments:monitor`

3.2.4 Wireshark

Build Platform	**pcwin-nt-x86 + Cygwin, pclin-2x-x86**
Release Type	**Developer**
Description	Wireshark is the tool used for decoding and analyzing logs generated by Synergy and BCHS. Installation and information on using Wireshark is available from: wireshark

3.3 Build Configuration

Customization of a CSR Synergy Framework build can be divided in two:

- **Build configuration**: This is configuration of generic build related parameters, which are unrelated to the functionalities of the Framework components. Examples include: debug enabling and log enabling.
- **Framework configuration**: This is configuration of functionality of specific Framework components. Examples include defines for the retransmission timer of the BCSP transport protocol.

This section describes details of the build configuration only. The build configuration is handled through files `config-<config-name>.mk` and `target-<target-name>.mk` located in the `$(FW_ROOT)/config` and `$(BSP_ROOT)/scripts` folders, respectively. The `target-<target-name>.mk` configuration file contains target specific parameters like default compiler, default target libraries, and default target compiler flags. These parameters are supposed to be fixed and chosen to be optimal for the given target platform. General parameters that change more frequently during development builds like enabling/disabling debug symbols, log generation, etc. are contained in the `config-<config-name>.mk` file.

3.3.1 General Build Parameters

The general build parameters are located in the `config-<config-name>.mk` configuration files. By default, the Framework build system uses configuration files where `<config_name> = default`. The default configuration can be changed using the `CONFIG` symbol as follows:

```
make all CONFIG=myconfig
```

The build configuration in `config-<config-name>.mk` consists of a simple tag-value list of parameters. The configuration parameters of `config-default.mk` are explained in Table 4.

Note: All configuration parameters can be overwritten from command line.

Configuration Parameter	Value
CONFIG	Is used to select a build configuration file as explained above. Defaults to CONFIG=default.
DEBUG	Enables debug friendly code generation, which, depending on the target, usually means inclusion of debugging symbols in the binary and possibly lowers the optimisation level. Default `DEBUG=1`.
LOG	Enables log output from all HCI communication and all interaction in the CSR Synergy Framework. `LOG=1` enables logging, `LOG=0` disables. Default `LOG=1`. **Note**: Synergy Technologies using the Framework Log API must be build against a Framework build where log is enabled.
USE_DEPENDENCY_FILES	Enables use of gcc dependency files – making build dependencies on changes to header (`.h`) files. `USE_DEPENDENCY_FILES=1` enables dependency files; 0 disables. Default `USE_DEPENDENCY_FILES=1`
ENABLE_SHUTDOWN	Enables calling of deinit function for Framework tasks (BCCMD, VM, HQ, TM). `ENABLE_SHUTDOWN=1` enables shutdown, `0` disables. Default `ENABLE_SHUTDOWN=1`.
EXCEPTION_PANIC	Enables all exception handler calls to end in a call to panic handler. `EXCEPTION_PANIC=1` enables, `0 disables`. Default `EXCEPTION_PANIC=0`.
DEVEL_TOOL_PATH	Developer tools
LOG_TOOL_PATH	Developer tools
CREATE_PRIM_CODE	Developer tools
FTS_VER	Version of FTS sniffer tool. Used when enabling FTS HCI log generation.
NUCLEUS_ROOT	The path to Nucleus PLUS RTOS – only needed when compiling to targets that run Nucleus PLUS RTOS. This parameter is not used for compiling generic code.
WIRESHARK_DIS_DEVEL_VERSION	Wireshark tool parameters. Developer tools.
WIRESHARK_BUILD_DIS_ROOT	Wireshark tool parameters. Developer tools.

Configuration Parameter	Value
WIRESHARK_VERSION	Wireshark tool parameters. Developer tools.
WIRESHARK_EXE_ROOT	Wireshark tool parameters. Developer tools.
WIRESHARK_PLUGIN_ROOT	Wireshark tool parameters. Developer tools.
CASTE_ROOT	Root to test framework. Developer tools.
CONFIG_H	Name of Framework configuration header. Default csr_usr_config_default.h
MBLK_DEBUG	When set to 1, enables certain debugging facilities in the csr_mblk components. Not recommended.
CSR_IP_SUPPORT_TLS	Controls whether TLS is implemented in the TLS or the IP task. If set to 1, the header files are configured to route TLS primitives to the IP task.

Table 4: config-default.mk

3.3.2 Target Build Parameters

The target specific build parameters are located in the target-<target-name>.mk configuration files. The file used is chosen by the TARGET symbol as target-$(TARGET).mk. The target build parameters included for the Framework reference targets pcwin-nt-x86, pclin-2.6-x86, and bdb2-nucleus-arm5tej are shown in Table 5, Table 6, and Table 7, respectively.

Note: All configuration parameters can be overwritten from command line.

Configuration Parameter	Description
CC	Compiler. Default CC=cl.
AR	Create archives (libraries). Default AR=lib
LD	Linker. Default LD=LINK
LDFLAGS	Linker flags. Default LDFLAGS=/nologo /machine:i386 /incremental:no /pdb:NONE /FIXED:NO /subsystem:console
DEFAULT_LIBS	Default libraries. Default DEFAULT_LIBS=Advapi32.
TARGET_CFLAGS	Compiler flags. Default TARGET_CFLAGS=/nologo /W3 /GX /GZ /D_MBCS /D_CONSOLE

Table 5: target-pcwin-nt-x86.mk

Configuration Parameter	Description
CC	Compiler. Default CC=gcc.
AR	Create archives (libraries). Default AR=ar

LD	Linker. Default LD=ld
DEFAULT_LIBS	Default libraries. Default DEFAULT_LIBS=pthread.
TARGET_CFLAGS	Compiler flags. Default TARGET_CFLAGS=-Wall -D_REENTRANT

Table 6: target-pclin-2.6-x86.mk

Configuration Parameter	Description
ARM_PATH	ARM RVCT compiler tool-chain path. Default ARM_PATH = "$(RVCT30BIN)"
ARM_COMPILER	ARM RVCT compiler. Default ARM_COMPILER=3.0
CC	Compiler. Default CC=$(ARM_PATH)/armcc
AR	Create archives (libraries). Default AR= $(ARM_PATH)/armar
LD	Linker. Default LD= $(ARM_PATH)/armlink
DEFAULT_LIBS	Default libraries. Default DEFAULT_LIBS= $(NUCLEUS_ROOT)/output/default/bdb2-nucleus-arm5tej/lib/nucleus.
TARGET_CFLAGS	Compiler flags. See file for default TARGET_CFLAGS

Table 7: target-bdb2-nucleus-arm5tej.mk

3.4 Building Libraries

As mentioned, the Framework build system is based on the GNU Make utility. Thus, a number of makefiles exist to build the libraries that constitute the CSR Synergy Framework software. The basic build command to build all Framework libraries is:

```
make all TARGET=<TARGET>
```

Libraries will be built using the configuration specified in the configuration files. Library naming and output will NOT reflect changes in configuration parameters inside the same configuration file. The output path of libraries will reflect the name of the config file. Thus, if libraries using different configurations are to exist at the same time, distinctly named configuration files must be used. However, if the need is only for e.g. enabling/disabling debug information or log generation, this will most likely be done by a recompilation using the same configuration file. The output path of libraries is distinct for GSP and BSP libraries, respectively. They are as follows:

- for GSP libraries: $(FW_ROOT)/output/<CONFIG_NAME>/<TARGET>/lib
- for BSP libraries: $(BSP_ROOT)/output/<CONFIG_NAME>/<TARGET>/lib

Libraries are built for each interface described in Section 2. All library names are prefix with csr_. A full list of library names for GSP and BSP libraries is provided in Table 8 and Table 9, respectively.

CSR Synergy Framework GSP Library	Description
csr_aclbuf_lower	ACL Buffering lower API

CSR Synergy Framework GSP Library	Description
csr_app	Application scaffolding library
csr_bccmd	BCCMD library
csr_bcsp	BCSP Transport Protocol library
csr_core_msg_converter	Log utility library
csr_data_store	Data store library
csr_data_store_lib	Data store utility library
csr_deprecated	Various deprecated wrapper functionality
csr_fastpipe	FastPipe library
csr_formatted_io	Formatted IO library
csr_fsal	File system abstraction layer task
csr_fsal_lib	File system abstraction task utility library
csr_sched	Scheduler
csr_h4ds	H4 Deep sleep library
csr_hci	Host Interface library
csr_hq	HQ library
csr_ip	TCP/IP network stack
csr_ip_ether_lib	TCP/IP network stack utility library
csr_ip_ifconfig_lib	TCP/IP network stack utility library
csr_ip_socket_lib	TCP/IP network stack utility library
csr_list	Linked list library
csr_log	Log scaffolding library
csr_log_btsnoop	Log library for FTS HCI log
csr_log_fts	Live FTS logging library
csr_log_pcap	Log library for Wireshark/PCAP log
csr_mblk	Zero copy data library
csr_message_queue	Message utility library
csr_msg_converter	Log utility library
csr_queue_lib	Message queue utility library

CSR Synergy Framework GSP Library	Description
csr_tm_bluecore	Transport Manager (TM) library
csr_sdio_cspi	CSR SDIO for CSPI
csr_sdio_sdio	CSR SDIO for SDIO
csr_type_a	Type-A Transport Protocol library
csr_type_a_async	Type-A Transport Protocol library using asynchronous CSR SDIO operations
csr_type_a_sync	Type-A Transport Protocol library using synchronous CSR SDIO operations
csr_ui	Application user interface scaffolding library
csr_usb	USB Transport Protocol library
csr_vm	VM library

Table 8: CSR Synergy Framework GSP Libraries

CSR Synergy Framework BSP Library	Description
csr_sched	Scheduler library
csr_time	Time library (System Time and UTC Time)
csr_panic	Panic library
csr_eh	Exception handling library
csr_logtrans	Log Transport library
csr_ser_com	UART library
csr_usb_com	USB library

Table 9: CSR Synergy Framework BSP Libraries

4 Document References

[SYN-FRW-SCHED-API]	CSR Synergy Framework Scheduler API. Doc. api-0004-sched
[SYN-FRW-TIME-API]	CSR Synergy Framework Time API. Doc. api-0020-time
[SYN-FRW-PANIC-API]	CSR Synergy Framework Panic API. Doc. api-0021-panic
[SYN-FRW-EXCEPTION-API]	CSR Synergy Framework Exception API. Doc. api-0022-exception
[SYN-FRW-UART-API]	CSR Synergy Framework UART API. Doc. api-0009-uart
[SYN-FRW-SDIO-API]	CSR Synergy Framework SDIO API. Doc. api-0012-sdio
[SYN-FRW-BCCMD-API]	CSR Synergy Framework BCCMD API. Doc. api-0003-bccmd
[SYN-FRW-HQ-API]	CSR Synergy Framework HQ API. Doc. api-0001-hq
[SYN-FRW-VM-API]	CSR Synergy Framework VM API. Doc. api-0002-vm
[SYN-FRW-FP-API]	CSR Synergy Framework FastPipe API. Doc. api-0010-fastpipe
[SYN-FRW-LOGTRANS-API]	CSR Synergy Framework Log Transport API. Doc. api-0008-log
[SPEC-TYPE-A]	SDIO Card Part E2, Type-A specification for Bluetooth

Terms and Definitions

Abbreviation	Explanation
BlueCore®	Group term for CSR's range of Bluetooth wireless technology chips
CSR	Cambridge Silicon Radio

Document History

Revision	Date	History
1	14 JUL 09	Initial version
2	19 OCT 09	Rename to COAL API to Scheduler API and move Time, Panic, and Exception APIs
3	30 NOV 09	Ready for release 2.0.0
4	20 APR 10	Ready for release 2.1.0
5	Dec 10	Ready for release 3.0.0
6	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.