UM1399
User manual

ST40 Micro Toolset

Introducing the ST40 Micro Toolset

The ST40 Micro Toolset is a cross-development system for developing and debugging C and C++ embedded applications on STMicroelectronics' ST40 range of products. All ST40 devices include the user debug interface (UDI), available through the JTAG port of the device, which provides on-chip emulation capabilities such as code and data breakpoints, watchpoints and memory peeking and poking.

The ST40 Micro Toolset provides an integrated set of tools to support the development of embedded applications.

This user manual provides detailed information to:

- enable users to run and debug code built for the ST40 family of processors on silicon and simulated targets
- enable users to customize and extend the support of the ST40 Micro Toolset for new hardware targets that use ST40 processors

Contents

Preface

Comments on this manual should be made by contacting your local STMicroelectronics sales office or distributor.

Document identification and control

Each book carries a unique identifier of the form:

nnnnnnn Rev *x*

where *nnnnnnn* is the document number, and *x* is the revision.

Whenever making comments on this document, quote the complete identification *nnnnnnn* Rev *x*.

License information

The ST40 Micro Toolset is based on a number of open source packages. Details of the licenses that cover all these packages can be found in the file `license.htm`. This file is located in the `doc` subdirectory and can be accessed from `index.htm`.

ST40 documentation suite

The ST40 documentation suite comprises the following volumes:

ST40 Micro Toolset user manual (7379953*)*

This manual describes the ST40 Micro Toolset and provides an introduction to OS21. It covers the various code and cross development tools that are provided in the ST40 Micro Toolset, how to boot OS21 applications from ROM and how to port applications which use STMicroelectronics OS20 operating systems. Information is also given on how to build the open source packages that provide the compiler tools, base run-time libraries and debug tools and how to set up an ST Micro Connect.

SH-4 generic and C specific ABI (7839242)

The SH-4 application binary interface (ABI) defines a system interface for application programs on SH-4 systems using the ELF executable and linking file format.

OS21 user manual (7358306)

This manual describes the generic use of OS21 across the supported platforms. It describes all the core features of OS21 and their use and details the OS21 function definitions. It also explains how OS21 differs from OS20, the API targeted at ST20 platforms.

OS21 for ST40 user manual (7358673)

This manual describes the use of OS21 on ST40 platforms. It describes how specific ST40 facilities are exploited by the OS21 API. It also describes the OS21 board support packages for ST40 platforms.

ST40 Micro Toolset GDB command scripts (8045872)

This document describes using GDB command scripts to connect to and configure a target board for loading and debugging programs through GDB.

32-Bit RISC series, ST40 Core and instruction set architecture manual (7182230)

This manual describes the architecture and instruction set of the ST40 core as used by STMicroelectronics.

ST40 core support peripherals manual (7988763)

This manual describes the ST40 core support peripheral (CSP) package that give the optional peripherals for use in ST40-based system-on-chips (SoCs).

ST Micro Connection Package documentation suite

The following documents are not distributed with the ST40 Micro Toolset, but can be obtained from your ST FAE or ST support center.

ST TargetPack user manual (8020851)

This manual describes the ST TargetPack, which is a method of describing target systems based upon ST system-on-chip devices.

Developing with an ST Micro Connect and ST TargetPacks application note (8174498)

This application note describes various aspects of using an ST Micro Connect host-target interface for system development.

Conventions used in this guide

General notation

The notation in this document uses the following conventions:

- `sample code`, `keyboard input` and `file names`
- *variables*, *`code variables`* and *`code comments`*
- `equations` and `math`
- **screens**, **windows**, **dialog boxes** and **tool names**
- **instructions**

Hardware notation

The following conventions are used for hardware notation:

- REGISTER NAMES and FIELD NAMES
- PIN NAMES and SIGNAL NAMES

Software notation

Syntax definitions are presented in a modified Backus-Naur Form (BNF) unless otherwise specified.

- Terminal strings of the language, that is those not built up by rules of the language, are printed in bold teletype font. For example, **`void`**.
- Non-terminal strings of the language, that is those built up by rules of the language, are printed in italic teletype font. For example, *`name`*.
- If a non-terminal string of the language is prefixed with a non-italicized part, it is equivalent to the same non-terminal string without that non-italicized part. For example, `vspace-`*`name`*.
- Each phrase definition is built up using a double colon and an equals sign to separate the two sides (‘`::=`’).
- Alternatives are separated by vertical bars (‘`|`’).
- Optional sequences are enclosed in square brackets (‘`[`’ and ‘`]`’).
- Items which may be repeated appear in braces (‘`{`’ and ‘`}`’).

Terminology

The first ST Micro Connect product was named the “ST Micro Connect”. With the introduction of the ST Micro Connect 2 and ST Micro Connect Lite, the original product is now known as the “ST Micro Connect 1” and the term “ST Micro Connect” refers to the family of ST Micro Connect devices. These names can be abbreviated to “STMC”, “STMC1”, “STMC2” and “STMCLite”.

Acknowledgements

SuperH® is a registered trademark for products originally developed by Hitachi, Ltd. and is owned by Renesas Technology Corp.

Microsoft®, MS-DOS®, Windows® and Windows Vista® are registered trademarks of Microsoft Corporation in the United States and other countries.

1 Toolset overview

The ST40 Micro Toolset is a cross-development system for developing and debugging C and C++ embedded applications on STMicroelectronics' ST40 range of products. All ST40 devices include the user debug interface (UDI), available through the JTAG port of the device, which provides on-chip emulation capabilities such as code and data breakpoints, watchpoints and memory peeking and poking.

The ST40 Micro Toolset provides an integrated set of tools to support the development of embedded applications.

1.1 Toolset features

- Supported host platforms
  The toolset is available on:
  - Microsoft Windows XP, Vista or 7
  - Red Hat Enterprise Linux 4 or 5 (or compatible)
- Code development tools (assembler, compiler and linker)
  Program development is supported by the GNU C compiler, the GNU C++ compiler, assembler, linker and archiver (librarian) tools.
- The ST40 simulator
  This provides an accurate software simulation of the entire family of STMicroelectronics' ST40 cores.
- Cross development with GDB
  The GNU debugger (GDB) supports both the ST40 simulator and the hardware development boards. GDB also includes a text user interface and the Insight GUI as a graphical user interface on all supported host platforms. The **sh4xrun** tool is also available to provide a command line driven interface to simplify downloading and running applications on the supported targets using GDB.
- STWorkbench Integrated Development Environment (IDE)
  The STWorkbench is built on the Eclipse IDE. The framework is extended using the CDT (C/C++ Development Tools) and ST40 specific plug-ins which provide a fully functional C and C++ IDE for STWorkbench. This allows the user to develop, execute and debug ST40 applications interactively. In addition, the ST Profiler, Coverage and Trace features enable profiling, coverage and trace data to be collected and analyzed.
- OS21 real-time kernel
  The software design of embedded systems is supported by a real-time kernel (OS21) which facilitates the decomposition of a design into a collection of communicating tasks and interrupt handlers.
- A C/C++ run-time system
  The **newlib** C library provides ANSI C run-time functions including support for C I/O using the facilities of the host system. The C++ run-time system is provided by the GNU GCC **libstdc++** library which includes support for the STL and **iostream** ISO C++ standard libraries.
- File I/O is provided as well as terminal I/O
- Trace and statistical data analysis tools
  The toolset supports tracing of OS21 kernel activity and OS21 API calls. The user may control OS21 Trace either by using GDB commands or function calls embedded in the

application. Trace and other statistical information can be viewed graphically in STWorkbench.

- Flash ROM examples
  Several Flash ROM examples are provided. These create applications which are able to boot from NOR Flash and from NAND Flash ROM on the supported targets.
- Support for the ST Micro Connect
  Provides the download route to the board through the JTAG interface. The ST Micro Connect supports download using Ethernet and USB from any host. The ST Micro Connect interface is connected to the UDI port of the target device, which is used to control and communicate with the device during development.
- ST TargetPacks
  ST TargetPacks are a method of describing target systems based on SoC devices. ST TargetPacks provide a single, definitive description of a target system for use by various tools within the development environment (such as **sh4xrun**).
- Profiling and coverage support
  Performance data can be obtained when running an application on an ST40 simulator and used to generate statistical and trace information. Performance data can also be acquired from an application running on a target board connected to an ST Micro Connect. The data can be analyzed using the STWorkbench or tools such as **sh4gprof** and **sh4gcov**.
- MTT (multi-target trace) library
  Provides a trace API to log applicative traces over the target's STM IP interface at both OS and user level.

The targets supported by this toolset are:

- STMicroelectronics development boards
  These boards provide development targets for the STMicroelectronics system-on-chip devices that use the ST40 core.
- ST40 simulator
  This provides an accurate simulation of SuperH architecture cores (such as the ST40) and comes in two forms.
  - Functional simulator
    Simulates the core accurately, but ignoring the internal details such as pipelines.
  - Performance simulator
    Simulates the core in detail in order to model the performance.
- GDB simulator
  GNU GDB has a built-in simulator target for the SH-4 family of cores. The GDB simulator is a user-mode-only basic instruction set simulator, with no support for memory management, executing privileged instructions, nor any of the extra facilities that the ST40 simulator provides.

Note: In the text of this document, the term ST40 simulator is used when referring to features that are common to both the functional and the performance simulators.

1.2 The SuperH configuration

The ST40 Micro Toolset, maintained by STMicroelectronics, is derived from the GNU tools maintained by the Free Software Foundation (FSF) and the open source community. The toolset provides a complete toolset supporting the ST40 core.

The traditional configuration (that is, the non-SuperH configuration) of the GNU tools for SH-4 is identified by the target triplet[a] **sh-elf**, or sometimes by the target triplet **sh-hitachi-elf** (which is an alias).

The SuperH configuration of the GNU tools is identified by the target triplet **sh-superh-elf**. The SuperH configuration is the only configuration supported by STMicroelectronics for the ST40 Micro Toolset.

1.2.1 Traditional and SuperH configuration differences

There are several changes that have been made to the traditional configuration to create the SuperH configuration.

- The default endianness has changed from big to little endian.
- Board support has been added. This allows the same tools to create executables for different target boards and ST40 simulators.
- Run-time support has been added. This allows the same tools to create executables for different operating systems and for different I/O interfaces (for example, debug link and simulator traps).
- ANSI C I/O over the debug link has been added. This is provided by the Data Transfer Format (DTF), a low-level communication mechanism.
- ST40-300 series cores, ST40-400 series cores and ST40-500 series cores support added.

[a] The GNU tools support many platforms, operating systems and vendor configurations. The tools are configured by means of a triplet, the second part of which is optional. As an example, STMicroelectronics uses the triplet **sh-superh-elf** to represent the SuperH configuration of the tools for the **sh** platform using generic **elf** files (as used by bare machines).

1.3 Distribution content

The ST40 Micro Toolset distribution includes tools, libraries, configuration scripts and examples.

1.3.1 Tools

From the GNU binutils package

sh4as	GNU assembler
sh4ld	GNU linker
sh4addr2line	Convert addresses into file names and line numbers
sh4ar	Create, modify, and extract from archives
sh4c++filt	Demangle encoded C++ symbols
sh4elfedit	Edit the header of ELF format files
sh4gprof	GNU profiler
sh4nm	List symbols from object files
sh4objcopy	Copy and translate object files
sh4objdump	Display information from object files
sh4ranlib	Generate index to archive contents
sh4readelf	Display the contents of ELF format files
sh4size	List file section sizes and total size
sh4strings	List printable strings from files
sh4strip	Discard symbols

From the GNU make package

make	GNU make
sh4make	GNU make

From the GNU GCC package

sh4c++	GNU C++ compiler
sh4cpp	GNU C/C++ preprocessor
sh4g++	GNU C++ compiler
sh4gcc	GNU C compiler
sh4gcov	GNU test coverage tool

From the GNU GDB/Insight package

sh4gdb	GNU target debugger
sh4insight	Graphical User Interface for the debugger
sh4run	GNU SH-4 simulator

Others

sh4xrun	SuperH target loader
censpect	Statistical data viewer
trcview	Trace data viewer
os21decodetrace	Decode tool for OS21 Trace
os21usertrace	User trace tool for OS21 Trace (implemented as a Perl script)
os21usertracegen	Tool to generate definition files for **os21usertrace**.
os21prof	OS21 profiler (implemented as a Perl script)
sh4rltool	Relocatable library tool (implemented as a Perl script)

Reference only

The distribution also provides the sh-superh-elf-*tool* versions of the tools listed above (see *Section 1.2 on page 16*) that are invoked by their sh4*tool* counterparts with additional features enabled. As a result, use the short forms of the tools in preference to their long form counterparts.

Table 1 lists the tools where the short form and long form versions differ in the features they provide.

Table 1. sh-superh-elf-tool short and long versions

Long name	Short name	Additional features
sh-superh-elf-gcc	sh4gcc	Provides board and run-time package support for the -mboard (see *Section 3.5 on page 40*) and -mruntime (see *Section 3.6 on page 46*) options.
sh-superh-elf-g++	sh4g++	
sh-superh-elf-c++	sh4g++	
sh-superh-elf-gdb	sh4gdb	Provides support for silicon and ST40 simulator targets.[1]
sh-superh-elf-insight	sh4insight	Sets the environment for the Insight GUI without any additional requirements on the user. set mem inaccessible-by-default is set to off (default is on).

1. It is possible to enable this support for the long forms of the tools using a customized .shgdbinit file (see *Section 4.2.2 on page 57*).

The sh-superh-elf-*tool* tools are located in the sh-superh-elf/bin subdirectory.

Passing arguments from environment variables

Each of the tools that have an associated sh-superh-elf-*tool* can accept command line arguments passed from environment variables. Each tool has its own specific environment variable, as listed in *Table 2*, which has the form SH4*tool*OPT. Where there are **+** signs in the tool name, these are replaced by X in the name of the environment variable.

Table 2. Tools that accept arguments from an environment variable

Tool	Environment variable
sh4addr2line	SH4ADDR2LINEOPT
sh4ar	SH4AROPT
sh4as	SH4ASOPT
sh4c++	SH4CXXOPT
sh4c++filt	SH4CXXFILTOPT
sh4cpp	SH4CPPOPT
sh4elfedit	SH4ELFEDIT
sh4g++	SH4GXXOPT
sh4gcc	SH4GCCOPT
sh4gcov	SH4GCOVOPT

Table 2. Tools that accept arguments from an environment variable (continued)

Tool	Environment variable
sh4gdb	`SH4GDBOPT`
sh4gprof	`SH4GPROFOPT`
sh4insight	`SH4INSIGHTOPT`
sh4ld	`SH4LDOPT`
sh4make	`SH4MAKEOPT`
sh4nm	`SH4NMOPT`
sh4objcopy	`SH4OBJCOPYOPT`
sh4objdump	`SH4OBJDUMPOPT`
sh4ranlib	`SH4RANLIBOPT`
sh4readelf	`SH4READELFOPT`
sh4run	`SH4RUNOPT`
sh4size	`SH4SIZEOPT`
sh4strings	`SH4STRINGSOPT`
sh4strip	`SH4STRIPOPT`
sh4xrun	`SH4XRUNOPT`

If an argument contains spaces, then the whole argument must be quoted (either single or double quotes can be used, but they must balance). For example:

Windows shell:

```
set SH4GDBOPT=-ex "break _SH_posix_Exit_r"
```

Bourne shell (or compatible)

```
SH4GDBOPT="-ex \"break _SH_posix_Exit_r\""
export SH4GDBOPT
```

1.3.2 Libraries

The libraries are supplied for each of the possible target configurations supported by GCC; one version for each permutation of the ST40-specific compiler options that affect code generation and for the Application Binary Interface (ABI), such as floating-point and endianness. Therefore, whichever combination of target configurations is chosen to compile a user program, a library with the same permutation (except for optimizations) exists and is automatically selected by the compiler driver.

From the newlib package

An ISO/ANSI C run-time library (**libc** and **libm**) and header files. The run-time libraries also provide support for low-level I/O and additional maths functions. The low-level I/O is implemented by the Data Transfer Format library (**libdtf**), see *Section 1.4.3: The data transfer format (DTF) library on page 24*.

The toolset provides an alternative I/O library (**libgloss**, see *Section 1.4.4 on page 24*) for building applications to run on the GNU GDB SH-4 simulator (**sh4run**) and also a run-time library (**libprofile**) to support profiling with **sh4gprof**.

From the GNU GCC package

The toolset provides compiler intrinsics libraries (**libgcc** and variants) and a run-time library **libgcov** to support code coverage with **sh4gcov**.

From the libstdc++ subpackage of the GNU GCC package

The toolset provides an ISO/ANSI C++ run-time library (**libstdc++**) and header files supporting I/O streams and the standard templates library (the **STL**).

Others

The OS21 real-time kernel library and header files, and OS21 board support libraries for the various supported platforms.

The relocatable loader library (**librl**) and header files.

The zlib compression library (**libz**) and header files.

The MTT (multi-target trace) library and header files.

1.3.3 Configuration scripts

A full set of GDB command scripts are supplied for connecting to and configuring the various supported platforms (used in conjunction with **sh4gdb**, **sh4insight** and **sh4xrun**).

A GDB command script (`.shgdbinit`) is provided in the subdirectory `sh-superh-elf/stdcmd` of the release installation directory to make these scripts available. This file is automatically read by **sh4gdb**, **sh4insight** and **sh4xrun**.

1.3.4 Sources

The package includes full sources for the OS21 real-time kernel library, OS21 Trace library and the OS21 profiler library. These are located in the `sh-superh-elf/src` directory.

1.3.5 Examples

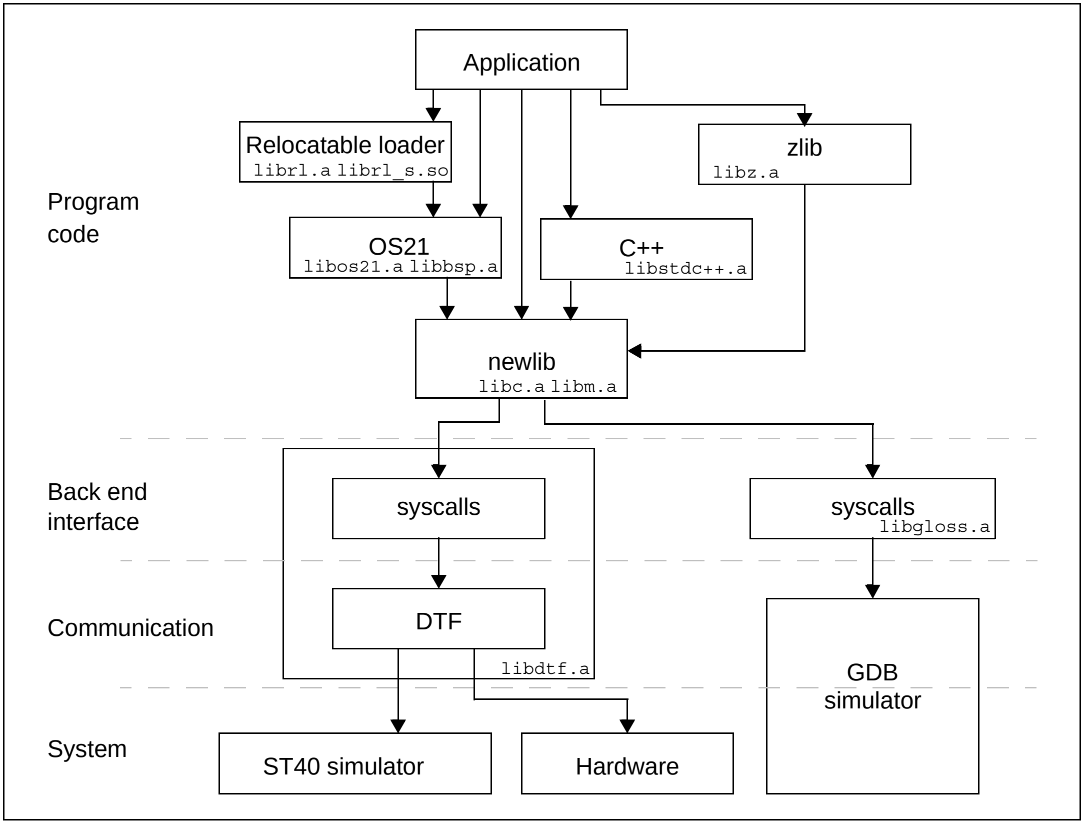
The toolset includes various example applications including those using OS21 and illustrating the construction of Flash ROM systems. See *Section 1.5.3: The examples directory on page 28* for more information on the examples supplied.

1.4 Libraries delivered

The ST40 Micro Toolset includes ANSI/ISO C and C++ run-time libraries and header files, supporting both OS21 and bare machine applications for various target application configurations.

Note: A “bare machine application” is a non-OS21 application built without real-time kernel libraries.

Figure 1. The relationship between the libraries



The header files shipped with the toolset are located in the subdirectory `sh-superh-elf/include` of the release installation directory and include the header files for OS21 support. The OS21 header files are located under `sh-superh-elf/include/os21`.

The libraries shipped with the toolset are located in the subdirectory `sh-superh-elf/lib` of the release installation directory, which is structured as described below.

- Little endian libraries with pervading double precision FPU support (the default, or optionally selected by the GCC compiler's `-ml` option):

  `sh-superh-elf/lib`
- Little endian libraries with no FPU support (selected by the GCC compiler's `-m4-nofpu` option and optionally the `-ml` option):

  `sh-superh-elf/lib/m4-nofpu`
- Little endian libraries with pervading single precision FPU support (selected by the GCC compiler's `-m4-single` option and optionally the `-ml` option):

  `sh-superh-elf/lib/m4-single`
- Little endian libraries with single precision only FPU support (SH-3e ABI) (selected by the GCC compiler's `-m4-single-only` option and optionally the `-ml` option):

  `sh-superh-elf/lib/m4-single-only`
- Big endian libraries[b] with pervading double precision FPU support (selected by the GCC compiler's `-mb` option):

  `sh-superh-elf/lib/mb`
- Big endian libraries[b] with no FPU support (selected by the GCC compiler's `-mb` and `-m4-nofpu` options):

  `sh-superh-elf/lib/mb/m4-nofpu`
- Big endian libraries[b] with pervading single precision FPU support (selected by the GCC compiler's `-mb` and `-m4-single` options):

  `sh-superh-elf/lib/mb/m4-single`
- Big endian libraries[b] with single precision only FPU support (SH-3e ABI) (selected by the GCC compiler's `-mb` and `-m4-single-only` options):

  `sh-superh-elf/lib/mb/m4-single-only`

1.4.1 The C library (newlib)

newlib implements a version of the C library that is suitable for use in embedded systems. **newlib** supports the most common functions used in C programs, but not the more specialized features available in standard operating systems, such as networking support.

*Note: Wide character support is not enabled in the supplied version of **newlib**.*

newlib assumes a minimal set of OS interface functions (the **syscalls** API). These provide all the I/O, entry and exit, and process control routines required by programs using **newlib**. The syscalls API is implemented either by the **libdtf** DTF library (for ST40 simulator or silicon) or by the **libgloss** library (for GDB simulator).

1.4.2 The C++ library (libstdc++)

The C++ library is part of GNU Compiler Collection and uses the underlying C library for its basic functionality.

[b] Some ST40 products might not be validated for use in big endian mode, please see the relevant silicon product documentation for details.

1.4.3 The data transfer format (DTF) library

The DTF library implements the POSIX I/O mechanism used with the ST Micro Connect or the ST40 simulator. It implements most of the basic file I/O features required by the C library. The I/O is performed using the debug link and requires the correct host side software to be present (this is handled automatically by the supplied GDB connection commands).

It is not usually necessary for applications to call the DTF library directly since this is handled by the **newlib** C library low-level I/O interface (see *Section 1.4.6: The syscalls low-level I/O interface on page 24*).

Note: 1 *The DTF library assumes that GDB is present in order to provide the I/O services for the target. If GDB is not present (for example, in boot-from-ROM systems) then the DTF library either returns an error (see below), the application waits indefinitely on an I/O transaction that never completes, or the application fails in some other way.*

In order to allow applications linked with the DTF library (the default) to operate correctly in these circumstances, the DTF library provides a mechanism for disabling communication with GDB. In the application, define the global variable `_SH_DEBUGGER_CONNECTED` *to* `0` *in the C namespace. For example:*

```
int _SH_DEBUGGER_CONNECTED = 0;
```

If set to 0, the DTF library returns an error and sets `errno` *to* `EIO`*.*

The Flash ROM examples do this automatically (see the `rombootram` *example* `readme.txt` *file for further details).*

2 *All interrupts are blocked whilst the target is communicating with the STMC (either sending a request or receiving a response). This means that I/O calls (such as* `printf()`*) may in certain circumstances cause the application to demonstrate unexpected behavior. All other application functionality is unaffected*

1.4.4 The libgloss library

libgloss is intended to be the **newlib** backend library (so called because it glosses over the system details). It implements the interface between **newlib** and the underlying system, whatever that may be, in order to allow access to I/O and other system resources via a standardized trap interface recognized by the GNU GDB simulator.

1.4.5 The zlib library

zlib implements the compression algorithms specified by Internet Task Force standards RFC 1950, RFC 1951 and RFC 1952. Further details can be found at the **zlib** website (*www.zlib.net*).

1.4.6 The syscalls low-level I/O interface

The **syscalls** low-level I/O interface consists of the following functions. These functions provide all the I/O, entry and exit, and process control routines that **newlib** requires. The functions are:

```
__setup_argv_and_call_main        _chmod            _chown
_close_r         _creat           _execv            _execve_r
_exit            _fork_r          _fstat_r          _getpid_r
_gettimeofday_r  _kill_r          _link_r           _lseek_r
```

```
_open_r          _pipe            _raise           _read_r
_rename_r        _readenv_r       _sbrk_r          _stat_r
_system_r        _times_r         _unlink_r        _utime
_wait_r          _write_r         isatty           truncate
```

There are two versions of this interface implemented.

- **DTF**
  This is for use with the ST40 simulator and the ST Micro Connect. Not all functions are implemented. See *Section 1.4.3*.
- **libgloss**
  This is for use with the GNU GDB simulator. Not all functions are implemented. See *Section 1.4.4*.

DTF provides four additional functions:

```
opendir          closedir         readdir          rewinddir
```

The **syscalls** example provided with the toolset (see *syscalls example on page 28*) contains minimal implementations of the functions. These versions are sufficient to compile, link and execute an application but the application cannot perform I/O or utilize any of the services that these functions provide until fully functioning versions have been provided.

The example implementation provides an overview of each function but for further information the POSIX standard should be used as a reference.

Note: It is not required for all functions to be implemented.

1.4.7 Threading

The C (**newlib**) and C++ (**libstdc++**) libraries both provide support for thread-safe operation (although by different mechanisms).

The C library ensures thread-safe operation by using per-task re-entrancy structures and guards around critical regions (as described in the **newlib** source documentation). If OS21 tasks are not being used then the C libraries use a single re-entrancy structure for the application and no guards whereas the OS21 versions use a re-entrancy structure per task and OS21 semaphores and mutexes as guards.

The C++ library relies on the generic GNU threading interface provided by the STMicroelectronics' implementation of the GNU Compiler Collection. The implementation of the generic threading interface in the GNU Compiler Collection only provides support for bare machine applications; the default implementation does not provide thread-safe operation (and hence the C++ library is not thread safe). However, the implementation of the generic threads interface provides a mechanism whereby the default implementation may be overridden as they are weakly defined. Therefore the generic GNU threading interface provides a technique for supporting different OS implementations without requiring a different GNU Compiler Collection for each OS.

The weakly defined functions exported by the generic GNU threads interface are listed here.

```
int __generic_gxx_active_p(void);
int __generic_gxx_once(__gthread_once_t *once, void (*func)(void));
int __generic_gxx_key_create(__gthread_key_t *key, void (*dtor)(void *));
int __generic_gxx_key_delete(__gthread_key_t key);
void * __generic_gxx_getspecific(__gthread_key_t key);
int __generic_gxx_setspecific(__gthread_key_t key, const void *ptr);
int __generic_gxx_mutex_destroy(__gthread_mutex_t *mutex) ;
void __generic_gxx_mutex_init(__gthread_mutex_t *mutex);
int __generic_gxx_mutex_lock(__gthread_mutex_t *mutex);
int __generic_gxx_mutex_trylock(__gthread_mutex_t *mutex);
int __generic_gxx_mutex_unlock(__gthread_mutex_t *mutex);
int __generic_gxx_recursive_mutex_destroy(__gthread_recursive_mutex_t *mutex) ;
void __generic_gxx_recursive_mutex_init (__gthread_recursive_mutex_t *mutex);
int __generic_gxx_recursive_mutex_lock (__gthread_recursive_mutex_t *mutex);
int __generic_gxx_recursive_mutex_trylock (__gthread_recursive_mutex_t *mutex);
int __generic_gxx_recursive_mutex_unlock (__gthread_recursive_mutex_t *mutex);
```

These functions are weakly defined in the GNU Compiler Collection and are implemented as no-op functions. The OS21 library replaces the GNU Compiler Collection definitions of these functions with its own definitions. These definitions are OS21 implementations to ensure thread safe operation of the generic GNU threading interface, and therefore the C++ library.

The `__generic_gxx_active_p` function returns a status value indicating whether threading is active (`1`) or not (`0`). All other functions return a status value which indicates either success (`1`), failure (`0`) or not supported (`-1`).

The GNU Compiler Collection generic threading interface types are defined as pointers to anonymous structures. The actual definitions of the types are determined by the implementations of the functions.

The functions correspond closely to equivalent functions in the POSIX standard for threads. Refer to the POSIX documentation for implementation details.

1.5 Release directories

Table 3 lists the directories of the installation. Some of these directories are described in more detail in the following sections.

As well as including the directories shown in *Table 3*, the release installation directory also includes the files `index.htm` (which can be used to navigate the documentation supplied with the installation) and `version.txt` (which gives the version number of the toolset release).

Table 3. The release directories

Directory	Contents
`bin`	The tools.
`doc`	The documentation set.
`include`	Header files.
`lib`	Library files.
`libexec`	C/C++ compiler executables.
`man`	man(1) manual pages.
`microprobe`	The ST Micro Connect target applications.
`share`	GDB GUI configuration files.
`sh-superh-elf/bin`	Subset of the binutils tools used by GCC.
`sh-superh-elf/examples`	Example applications.

Table 3. The release directories

Directory	Contents
sh-superh-elf/include	OS21 and C/C++ library header files.
sh-superh-elf/lib	Run-time library files.
sh-superh-elf/src	Source files for OS21 and other packages.
sh-superh-elf/stdcmd	GDB command scripts.

1.5.1 GDB command scripts directory

The directory sh-superh-elf/stdcmd contains GDB command scripts that define connection commands for simulators and for target boards supplied by STMicroelectronics. See the *GDB command scripts user manual* (8045872) for information about these command scripts.

1.5.2 The documents directory

Several HTML files are provided to navigate the documentation. These can all be accessed from the index.htm file in the release installation directory. *Table 4* lists the main pages.

Table 4. The HTML files in the doc directory

File	Description
acknow.htm	The acknowledgments page.
cdmap.htm	A map of the information provided.
docbug.htm	Instructions on how to get support on the toolset and report problems in the documentation.
docs.htm	A list of the documentation supplied with the toolset. Each document can be accessed from this page by clicking on the relevant link.
issues.htm	Information on bugs outstanding and resolved in this release.
license.htm	Links to each of the license files that the software is shipped under.

The doc directory also contains the supporting documentation supplied with the toolset. There are three subdirectories provided in the doc directory:

Table 5. The doc subdirectories

Directory	Description
images	The images used in the documentation.
hyper	The documentation in HTML format. These can be accessed from the docs.htm file.
pdf	The printable documentation supplied as one PDF file per document. These can be accessed from the docs.htm file.

1.5.3 The examples directory

The examples are located in the `sh-superh-elf/examples` directory. Each example has a `readme.txt` file describing the example and makefiles to build the example.

Also supplied is a sample implementation of the low-level I/O interface provided in the **libdtf** and **libgloss** libraries.

Getting started examples

The `bare/getstart` subdirectory of the `examples` directory contains two simple examples of bare machine programs. These examples can be used as confidence tests for the hardware and the toolset. There is a simple “Hello World” application, and one to display part of the Fibonacci sequence.

Hardware configuration registers

The `bare/sh4reg` subdirectory of the `examples` directory provides C/C++ header files. These header files define the locations of the memory mapped configuration registers for the core and other commonly accessed peripherals of the ST40.

MTT trace example

The `bare/mtt` subdirectory of the `examples` directory contains a “Hello World” example using MTT trace API calls.

syscalls example

The `syscalls` subdirectory of the `examples` directory contains a sample implementation of the **syscalls** low level I/O interface (see *Section 1.4.6: The syscalls low-level I/O interface on page 24*).

Census example

The `census` subdirectory of the `examples` directory contains the implementation of the API that enables an application to control performance data collection on a simulated ST40 from within the source (see *Section 8.3.5: Dynamic control on page 109*).

OS21 examples

The `os21` subdirectory of the `examples` directory contains some examples of programs using the features of OS21.

- The `os21/autostart` subdirectory contains an example of how to start OS21 before any C++ static constructors or `main()` are called.
- The `os21/dynamic` subdirectory contains an example illustrating how to build a simple application which loads a dynamic library from the host file system.
- The `os21/failsafe` subdirectory contains an example which runs a fail-safe (that is, integrity checking and, potentially, repairing) application before running the normal boot application.
- The `os21/lowpower` subdirectory contains an example showing how to use the OS21 power management API to put an ST40 core into low power standby mode.
- The `os21/mandelbrot` subdirectory contains a multi-tasking example generating a Mandelbrot pattern.

- The `os21/mtt` subdirectory contains a "Hello World" and Mandelbrot example using MTT trace API calls for OS21.
- The `os21/nandboot` and `os21/nandbootblock0` subdirectories contain an example of booting from NAND Flash ROM using an ST NAND controller, loading an application into RAM and executing it. The `nandboot` example requires an ST NAND controller that supports multiple block remapping in boot mode, whilst the `nandbootblock0` example is for older ST NAND controllers that do not support multiple block remapping in boot mode.
- The `os21/os21demo` subdirectory contains an example of using tasks to animate a graphical display.
- The `os21/rombootram` and `os21/rombootrom` subdirectories contains examples of how to program Flash ROM, and how to build applications which can be booted from Flash ROM.
- The `os21/rombootanywhere` subdirectory contains an example that has some of its code and data placed in an alternative memory location.
- The `os21/romdynamic` subdirectory shows how to use the Relocatable Loader Library to load a dynamic library from Flash ROM from an application which is booted from Flash ROM.
- The `os21/romloader` subdirectory contains an example showing how the ST40 core can boot the ST40 and ST231 slave cores of an SoC. This example requires the ST200 Micro Toolset R4.1 or later.
- The `os21/rommultiboot` subdirectory contains an example showing how the ST40 and ST231 cores of an SoC can boot from Flash ROM. This example requires the ST200 Micro Toolset R4.1 or later.
- The `os21/sh4ubc` subdirectory contains an example illustrating using the ST40 UBC to perform run-time debugging of an application without the use of a debugger (see the *ST40 Core Architecture Manual*). The example contains the source for the UBC library and a small test program that uses the library.
- The `os21/soaktest` subdirectory contains a simple stress test program, designed to act as a confidence test for OS21 running on the target.
- The `os21/sti5528dualboot` subdirectory contains an example showing how both the ST40 and ST20 cores on the STi5528 device can boot from Flash ROM. This example requires the ST20 Embedded Toolset R1.9.6 patch 7 or later.
- The `os21/sti5528loader` subdirectory contains an example showing how the ST40 core on the STi5528 device can boot the ST20 core. This example requires the ST20 Embedded Toolset R1.9.6 patch 7 or later.
- The `os21/timer` subdirectory contains an example showing how the OS21 API can be used to implement a simple timer. Tasks are able to create timer objects, which have a programmable duration, and can run in one shot or periodic mode. When a timer fires, a user supplied callback function is called in the context of a high priority task. The example contains the source for the timer library, and a small test program which uses the library.

2 Introducing OS21

OS21 is a royalty-free, lightweight, multitasking operating system developed by STMicroelectronics. It is an evolution of the OS20 API and is intended for applications where small footprint and excellent real-time responsiveness are required. It provides a multi-priority preemptive scheduler, with low context switch and interrupt handling latencies.

OS21 assumes an unprotected, single address-space model and is designed to be easily portable between SoC architectures.

OS21 provides an OS20 compatible API to handle task, memory, messaging, synchronization and time management. In addition, OS21 enhances the OS20 memory API and introduces API extensions to control virtual memory, mutexes, event flags, interrupts and caches.

OS21 aware debugging is available through GDB.

Multi-tasking is widely accepted as an optimal method of implementing real-time systems. Applications may be broken down into a number of independent tasks which co-ordinate their use of shared system resources, such as memory and CPU time. External events arriving from peripheral devices are made known to the system via interrupts.

The OS21 real-time kernel provides comprehensive multi-tasking services. Tasks synchronize their activities and communicate with each other via semaphores, event flags, mutexes and message queues. Real world events are handled using interrupt routines and communicated to tasks using semaphores and event flags. Memory allocation for tasks is selectively managed by OS21, the C run-time library or the user. Tasks may be given priorities and are scheduled accordingly. Timer functions are provided to implement time and delay functions.

An OS21 application is built as a single executable image[a], which can be loaded on the target through a debug link, or from Flash ROM. This single executable is typically written in C, and statically linked with the C run-time library, the OS21 library and the OS21 board support library. The application author has control of initializing the OS21 kernel, and switching on pre-emptive multi-tasking support. Once the OS21 kernel has been started, the full OS21 API can be used.

[a] This executable may load relocatable libraries. See *Chapter 11: Relocatable loader library on page 139*.

A very simple OS21 application (`test.c`) is shown below:

```
#include <os21.h>
#include <stdio.h>

void my_task (char *message)
{
  printf("Hello from the child task.\nMessage is '%s'\n", message);
}

int main (void)
{
  task_t *task;

  kernel_initialize(NULL);
  kernel_start();

  printf("Hello from the root task\n");

  task = task_create((void (*)(void*))my_task,
                     "Hi ya!",
                     OS21_DEF_MIN_STACK_SIZE,
                     MAX_USER_PRIORITY,
                     "my_task",
                     0);

  task_wait(&task, 1, TIMEOUT_INFINITY);

  printf("All tasks ended. Bye.\n");

  return 0;
}
```

To compile and run this program on an STb7100-MBoard connected to an ST Micro Connect called `stmc`:

```
sh4gcc test.c -mruntime=os21 -mboard=mb411
sh4xrun -t stmc -c mb411 -e a.out
```

The output should be:

```
Hello from the root task
Hello from the child task.
Message is 'Hi ya!'
All tasks ended. Bye.
```

For more information on OS21, see the *OS21 User Manual* (7358306) and the *OS21 for ST40 User Manual* (7358673).

2.1 OS21 features

The following list summarizes the key features of OS21.

- OS21 is a simple, royalty free multi-tasking package.
- There is a single address space and single name space (application has one executable image).
- There is a 256 level priority based FIFO scheduler.
- It has optional timeslicing.
- It has inter-task synchronization.
- Counting semaphores:
  - can be initialized to any count
  - can be signalled from interrupts
  - for FIFO semaphores, the longest waiting task gets the semaphore
  - for priority semaphores, the highest priority task gets the semaphore
- Mutexes can:
  - create critical sections between tasks
  - can be recursively acquired by the owning task without deadlock
  - for FIFO mutexes, the longest waiting task gets the mutex
  - for priority mutexes, the highest priority task gets the mutex and supports priority inheritance to avoid priority inversion
- Event flags where:
  - tasks can poll, or wait for all or any event flag within a group
  - events can be posted from a task or interrupt
- There is inter-task communication that uses simple FIFO message queues.
- There are user-installable interrupt handlers.
- There are user installable exception handlers.
- It has extensive cache API.
- The memory management has:
  - heaps
  - fixed block allocator
  - simple (non-freeable) allocator
  - user definable allocators
  - system heap managed by OS21 or C run-time
- It has virtual memory management.
- There is task aware profiling. The OS21 profiler allows profiling of a single task, a single interrupt level or the system as a whole.
- OS21 provides different power levels and a mechanism for transitioning between power levels, including waking up from standby mode.
- The board support package (BSP) libraries allow customization for new boards.
- OS21 uses the **newlib** C run-time library

2.2 OSPlus

OSPlus is an optional enhancement package for the OS21 operating system, and is available separately from the ST40 Micro Toolset. OSPlus provides a number of additional APIs that support the following features.

- Device driver infrastructure:
  - device driver API
  - device driver writer’s interface
  - registry services
  - drivers for standard devices
- File system infrastructure:
  - top level virtual file system (VFS)
  - support for FAT and Ext2 file systems
- Optional OSPlus extension packages:
  - USB 2.0
  - TCP/IP

For more information about OSPlus, see *OSPlus datasheet* (7813502).

3 Code development tools

The code development tools are based on the GNU development tools and have been modified in various ways from the standard GNU base tools. These modifications are referred to as the SuperH configuration (see *Section 1.2 on page 16*). These changes specialize the tools for the ST40 and provide integration with the ST40 simulators and ST40 target boards.

3.1 The GNU compiler (GCC)

GCC is the GNU Compiler Collection (formerly called the GNU C Compiler). It has support for a number of languages including C, C++, Objective C, Fortran, Ada and Java. Only the C and C++ compilers are provided and supported by STMicroelectronics.

The compiler tools are:

- **sh4gcc**
  The C compiler.
- **sh4g++**
  The C++ compiler.
- **sh4c++**
  An alternative name for the C++ compiler.

Note: These tools are compiler drivers. The actual compilers are called ***cc1*** *and* ***cc1plus*** *and are invoked by the compiler drivers for each C and C++ file. The compiler driver may also invoke the assembler and linker tools as necessary.*

3.1.1 GCC command line quick reference

Table 6 lists many of the most useful, generic, options to the compiler driver (which may also call the assembler and linker).

Table 6. sh4gcc command line quick reference

Option	Description
`-o` *`file`*	Use *`file`* as the output file.
`-c`	Compile and assemble only - no linking.
`-l`*`library`*	Link against *`library`*.
`-L` *`directory`*	Search *`directory`* for libraries.
`-I` *`directory`*	Search *`directory`* for header files.
`-isystem` *`directory`*	Search *`directory`* for system header files (included with `<>` not `""`).
`-S`	Do not assemble. Generate `.s` files containing assembly code instead.

Table 6. sh4gcc command line quick reference (continued)

Option	Description
`-E`	Preprocess only. Output preprocessed file to standard output or the file specified by `-o` if supplied. When generating preprocessed output, it is sometimes useful to dump all macro definitions using the `-dD` option. Further, comments can be preserved using the `-C` or `-CC` options.
`-save-temps`	Do full compile unless otherwise directed, but do not remove intermediate files. Preprocessed C output to `.i` files, preprocessed C++ output to `.ii` files, assembler code to `.s` files and object code to `.o` files.
`-Wa,`*`optionlist`*	Pass options to the assembler. Use commas instead of spaces within the option list (or use quotes).
`-Wp,`*`optionlist`*	Pass options to the preprocessor.
`-Wl,`*`optionlist`*	Pass options to the linker.
`-Wl,-Map,`*`file`*	Generate a linker map file whose name is *`file`*.
`-v`	Verbose output mode. This is useful for viewing the programs the compiler driver is invoking. If given as the first parameter with a short form of the tool, any additional options and environment variables set are also displayed.
`--help`	Provide help on the command line options.
`--help -v`	Provide help on all the options available on all the programs which the compiler driver may invoke (for example, the assembler and linker).
`--help=optimizer`	Provide help on the optimizer options.
`--help=target`	Provide help on the target options.
`-g`	Insert debug information into the output files.
`--coverage`	Enable program coverage.
`-pg`	Enable function profiling.
`-O0`	Do not optimize the output code. This is the default optimization setting if `-O` is not specified.
`-O1`	Do some optimizations. This is the default optimization setting if `-O` is specified without a level. Some optimizations enabled by `-O1` can reduce the ease of debugging.
`-O2`	Do most optimizations. Some optimizations enabled by `-O2` can severely impact the ease of debugging.
`-O3`	Do all `-O2` optimizations, plus additional aggressive optimizations that can increase code size.
`-Os`	Do optimizations designed to reduce code size (and skip optimizations that often increase code size).

Table 6. sh4gcc command line quick reference (continued)

Option	Description
`-flto`	Enable the link time optimizer. To be effective, use this option consistently for all of the compile, assemble and link steps. See *B.1.5: Link time optimization on page 253* for further information.
`-funroll-loops`	Enable loop unrolling; not enabled by default with `-O1` or `-O2`; enabled in `-O3`. Only unroll when the iteration count is known.
`-funroll-all-loops`	Enable loop unrolling for all loops; not enabled by default with `-O1`, `-O2` or `-O3`.
`-fomit-frame-pointer`	Use the offset from the stack pointer instead of a frame pointer to access frame variables. This frees R14 for other uses. This is enabled by default.
`-Wall`	Give all but the most unusual warnings.
`-pedantic`	Give all warnings required by the ISO C standard.
`-D`*`macro`*`[=`*`value`*`]`	Define a preprocessor *`macro`* (same as `#define`). If `=`*`value`* is not given then the default is `1`.

Table 7 lists some generic options that are common to all the SH-4 core families and do not rely on the SuperH configuration.

Table 7. sh4gcc SH-4 specific options

Option	Description
`-ml`	Compile for a little endian target (default for SuperH and ST40 cores).
`-mb`	Compile for a big endian target.
`-mrelax`	Do special linker optimizations. To be effective, use consistently for all of the compile, assemble and link steps. See *B.1.6: Stack overflow checking on page 256* for further information.
`-m4-300`	Compile for an ST40-300 series core.

For more SH-4 specific options, see *Table 56: SH-4 specific GCC options on page 247*. For ST40 specific optimization recommendations, see *Section B.1.10: Speed and space optimization options on page 261*.

3.1.2 GCC SuperH configuration specific options

Table 8 lists all the options added by the SuperH configuration.

Table 8. sh4gcc SuperH configuration specific options

Option	Description
`-mboard=`*`board`*	This option is used by the board support package. The board support package allows the same toolchain to build executables for a number of different hardware and simulation platforms (not including OS platforms). This option **must** be specified when linking. The value of *`board`* determines the memory location at which the linker places the program and the stack; therefore this value determines the boards on which the application will run. For a list of the valid values, see the HTML document `doc/boardsup.htm` in the release installation directory.
`-mruntime=`*`runtime`*	This option is used by the run-time support package to select the I/O interface and to select between bare machine and OS21 applications. The default is to compile for a bare machine application using DTF. For a list of the valid values, see *Section 3.6: Run-time support on page 46*. For instructions on setting up a custom run-time, see *Section 3.6.1: GCC run-time support setup on page 47*.
`-profiler`	Use this option to enable dynamic OS21 profiler support. See *Chapter 13: Dynamic OS21 profiling on page 214*.
`-profiler-specs=`*`file`*	Use this option to replace the default OS21 profiler specs file `os21profiler.specs` with *`file`*.
`-profiler-no-constructor`	Use this option to disable the automatic initialization of the dynamic OS21 profiler.
`-rlib`	Use this option to build a relocatable library. See *Section 11.5: Writing and building a relocatable library and main module on page 156*.
`-rmain`	Use this option to build an executable which can load relocatable libraries. See *Section 11.5: Writing and building a relocatable library and main module on page 156*.
`-trace`	Use this option to enable OS21 Trace support. See *Chapter Appendix D: Profiler plugin on page 270*.
`-trace-api`	Use this option to enable OS21 API trace support.
`-trace-specs=`*`file`*	Use this option to replace the default OS21 trace specs file `os21trace.specs` with *`file.`*
`-trace-api-`*`class`*	Use this option to include the specified *`class`* of API in the tracing.
`-trace-api-no-`*`class`*	Use this option in conjunction with the `-trace-api` option to exclude API functions from the specified *class* of API from tracing.

Table 8. sh4gcc SuperH configuration specific options (continued)

Option	Description
`-trace-no-constructor`	Use this option to disable the automatic initialization of the OS21 Trace buffers.
`-trace-no-destructor`	Use this option to disable the OS21 Trace destructors on application exit.

3.2 The GNU assembler

It is not usually necessary to invoke the assembler directly since the GNU compiler driver calls the assembler automatically when an assembler source file is specified. However, there may be occasions when it is necessary to invoke the assembler directly. The assembler tool is **sh4as**.

3.2.1 GNU assembler command line quick reference

Table 9 lists the most useful options to the assembler.

Table 9. GNU assembler command line quick reference

Option	Description
`-little`	Assemble for a little endian target. This is the default setting.
`-big`	Assemble for a big endian target.
`-o` *`file`*	Use *`file`* as the name of the output file. The default is `a.out`.
`-relax`	Place special information in the object file, allowing some additional optimizations to be performed by the linker. This is only useful if the relaxation options of the compiler and linker are also used. The `-mrelax` option to the compiler driver does this automatically (if the compile, assemble and link steps are performed as a single operation).

3.3 The GNU linker

As with the assembler, it is generally unnecessary to use the linker directly since the GCC compiler driver calls it automatically (using the GCC **collect2** tool). The linker tool is **sh4ld**.

The SuperH configuration is different from the traditional configuration in the set of supported emulations. These emulations support board support packages (see *Section 3.5: Board support on page 40*).

3.3.1 GNU linker command line quick reference

Table 10 lists many of the most useful linker options.

Table 10. sh4ld command line quick reference

Option	Description
`-l`*`library`*	Specify a library. The linker searches its search path for a file named `lib`*`library`*`.a`. Only the first one found on the path is used.
`-L` *`path`*	Add *`path`* to the library search path.
`-m` *`emulation`*	Use *`emulation`* to link the files. The emulation selects a linker script file from the standard set.
`-EL`	Link for a little endian target. Outside of the SuperH configuration a little endian emulation is also required. (Default for SuperH configuration.)
`-EB`	Link for a big endian target. Outside of the SuperH configuration a big endian emulation is also required. (Default for traditional configuration.)
`-t`	Output trace information for the link process. This allows the dependencies to be traced, which is useful for diagnosing link failures.
`-T` *`file`*	Provide a custom linker script file. This overrides the linker script from the emulation.
`-r`	Create a relocatable object file as output. Used for partial links.
`-M`	Output map information from the link to standard output.
`-Map` *`file`*	Output map information from the link to *`file`*.
`-s`	Strip all symbols from the output. Reduces the output file size, but cannot be debugged.
`-S`	Strip debugging symbols from the output.
`--relax`	Do relaxation optimizations (requires that the inputs are compiled and assembled with the relaxation options).
`--defsym` *`s`*`=`*`v`*	Define symbol *`s`* to value *`v`*. This option is used by the board support mechanism to set the top and bottom of memory, the program start and the stack location.

3.4 Profiling with the sh4gcov and sh4gprof utilities

sh4gcov is used for testing the coverage of a program. This has two main purposes:

- to identify how much of the program code has been tested
- to identify the most-used parts of the program

To use **sh4gcov**, the entire application must first be instrumented by compiling with the `--coverage` compiler option.

When the application is executed a file named *`program`*`.gcda` is created in the same directory as the object file, for each source file compiled using the `--coverage` compiler option. This file can then be read by **sh4gcov**:

```
sh4gcov program.c
```

sh4gcov creates a file named *`program`*`.c.gcov` which records the number of times each line was executed. Lines that were not executed are marked with `####`. The `-f` option provides a summary-per-function to the console:

```
sh4gcov -f program.c
```

The counts are cumulative. Therefore if the application is run multiple times, the execution counts in *`program`*`.gcda` are added to those of previous runs. This allows testing of all possible paths through the application.

The `-b` option provides information about how many times each branch was taken:

```
sh4gcov -b program.c
```

This provides a summary to the console and also records specific information in the `.gcov` file.

sh4gprof is used for profiling the application. For best results, the entire application should be instrumented by compiling with the `-pg` option.

When the application is executed the file named `gmon.out` is created which provides the profiling information. **sh4gprof** can then be used to examine this data.

There are several options for viewing different aspects of the data. Extensive support for profiling and coverage analysis is provided in the STWorkbench IDE, see *Chapter 5: Using STWorkbench on page 68*. Also refer to the GNU *Using the GNU Compiler Collection* manual for more information.

3.5 Board support

The board support mechanism specifies the memory map that is used to link the program. This defines the top and bottom of memory, the program start and the stack location.

The default memory layout places the data and text (code) sections of the program at the lowest available address and places the stack at the highest available address. The heap is placed after the data and text sections and grows towards the stack.

The set of board specifications for a particular release is contained within the `boardspecs` file (see *Section 3.5.1: GCC board support setup*) and is selected with the `-mboard` GCC option. For example, the option `-mboard=mb411` selects the memory map for the STb7100-MBoard.

A suitable connect command or ROM loader is required to configure the target. For example, the **sh4gdb** GDB command `mb411` connects to the target and configures it so that it is suitable for a program linked for the STb7100-MBoard. See *ST40 Micro Toolset GDB command scripts* (8045872) for more information about GDB connection commands.

The ST40 has two physical addressing models; 29-bit and 32-bit mode (also known as Space Enhancement mode, or SE mode for short). SE mode is only supported on some variants of the ST40. Further detailed information on these modes can be found in the *ST40 Core Architecture Manual* (7182230).

The ST40 memory space is divided into 5 memory regions. Each region has different cache and translation properties. *Table 11* lists the memory regions and their properties. See *ST40 Core Architecture Manual* (7182230) for further details.

Table 11. ST40 memory regions

Region	Address range	Description

P0	`0x00000000`
	`0x7FFFFFFF`
		Cacheability and address translation is controlled by the Unified Translation Look-aside Buffers (UTLBs) when the Memory Management Unit (MMU) is enabled.
		If the MMU is not enabled, then cacheability is controlled by the ST40 cache control register (CCN.CCR). Physical addresses are translated either by an identity mapping in SE mode or by masking the address to 29 bits.

P1	`0x80000000`
	`0x9FFFFFFF`
		Cacheability and address translation is controlled by the Privileged Mode Buffers (PMBs) when in SE mode.
		If SE mode is not enabled then cacheability is controlled by the ST40 cache control register and the address is masked to 29 bits.

P2	`0xA0000000`
	`0xBFFFFFFF`
		Cache and address translation is controlled by the Privileged Mode Buffers (PMBs) when in SE mode.
		If SE mode is not enabled then the memory region is uncached and the address is masked to 29 bits.

P3	`0xC0000000`
	`0xDFFFFFFF`
		Same behavior as for P0.

P4	`0xE0000000`
	`0xFFFFFFFF`
		Special region which is uncached with no address translation.
		This region contains resources such as memory mapped control registers for the ST40 and peripheral registers.

29-bit, non SE-mode applications are linked by default to the P1 (cached) region. A different region can be specified by adding the appropriate `px` suffix (where *`x`* is the region number) to the board specification name.

Note: *STMicroelectronics recommend that applications do not modify the top three address bits to obtain uncached views of physical memory. The method is not portable and works only with applications that are linked for non SE-mode or with the `se29p1` (or `se29p2`) `-mboard` board name suffixes. Use the OS21 virtual memory API to obtain uncached views of physical memory. For an example, see Section A.5: Access to uncached memory on page 230.*

SE-mode applications are supported by the following -mboard board name suffix variants:

se The se board specification suffix assumes that the PMBs are configured so that the external RAM is mapped at the base of the P1 region. If there are two banks of RAM (such as on the STb7109 and STi7200) then the second bank is mapped at the base of the P2 region.

The se board suffix is the standard -mboard suffix to be used when linking applications to execute in SE mode.

se29p1 The se29p1 board specification suffix assumes that the PMBs are configured so that all banks of external RAM are mapped contiguously from the base of P1 with the caches enabled, and with the same mapping from the base of P2 but with the caches disabled. The se29p1 suffix links applications for the cached mapping of RAM.

This configuration provides a 29-bit physical addressing compatibility mode that enables the top three bits of the P1 and P2 addresses to be modified in order to switch between cached and uncached views of physical memory.

se29p2 The se29p2 board specification suffix assumes the same PMB configuration as the se29p1 suffix and links applications for the uncached mapping of RAM.

Note: Support for 32-bit memory is provided using the OS21 virtual memory API, see OS21 User Manual (7358306) and OS21 for ST40 User Manual (7358673).

For ROM examples in the release that support 32-bit memory, see *Chapter 10: Booting OS21 from Flash ROM on page 135*.

Please refer to the HTML document doc/boardsup.htm in the release installation directory for the complete list of -mboard options recognized by GCC for each supported SoC.

3.5.1 GCC board support setup

The GCC -mboard option is controlled by the GCC specs file boardspecs. This file is located in the sh-superh-elf/lib/gccscripts subdirectory of the release installation directory. A description of the format of a GCC specs file may be found in *Section 3.15* of the *Using the GNU Compiler Collection* manual.

The primary objectives of the boardspecs file are to provide the linker with the following:

- the start address for placing the linked executable (using the _start symbol)
- the address of the top of the stack (using the _stack symbol) for the given board

For the -mboard options defined by the default boardspecs file, when linking for 29-bit, non SE-mode, the addresses for these symbols are derived from ___rambase and ___ramsize, which specify the location and size of RAM allocated to the ST40 core. Similarly ___rombase and ___romsize specify the location and size of the ROM from which the ST40 core boots. The symbol .reservedramsize defines the size of reserved memory at the base of RAM and is used for calculating the start address.

See *Section 3.5.2: Linker board support on page 45* for details of the linker command line.

Note: ___rambase and ___rombase are board-specific and are derived from .physrambase and .physrombase as shown in the example below. Specifically, ___rambase always refers to the virtual address of .physrambase (minus the value of .rambaseoffset, if defined).

Refer to the `boardspecs` file for details of how the `_start` and `_stack` symbols are defined when linking using the SE mode `-mboard` options.

The `boardspecs` file works by defining a spec string named `board_link`. This spec string must, directly or indirectly, specify the linker options defining the memory available to the application for the target board.

An example of the simplest `boardspecs` file is as follows:

```
*board_link:
--defsym _start=0xA4001000 --defsym _stack=0xA5FFFFFC
```

This defines the memory for an STb7100-MBoard in the P2 region. The first line (and it **must** be the first line) describes that it is defining or re-defining the `board_link` spec-string. The second line (and it **must** be the second line) is the definition of the spec-string. The string is inserted into the linker command line when the spec-strings are interpreted by the GNU GCC compiler driver.

The following examples show the linking of applications that rely on the `___rambase` and `___ramsize` symbols (or their ROM counterparts). The examples directly define the memory for a single board with no option to support any other board.

```
*start_and_stack:
--defsym _start=___rambase+.reservedramsize \
--defsym _stack=___rambase+___ramsize-4

*board_link:
--defsym .reservedramsize=0x1000 \
--defsym ___rambase=0xA4000000 --defsym ___ramsize=0x02000000 \
--defsym ___rombase=0xA0000000 --defsym ___romsize=0x00800000 \
%(start_and_stack)
```

An example of defining the memory indirectly is as follows:

```
*start_and_stack:
--defsym _start=___rambase+.reservedramsize \
--defsym _stack=___rambase+___ramsize-4

*mb411:
--defsym .reservedramsize=0x1000 \
--defsym ___rambase=0xA4000000 --defsym ___ramsize=0x02000000 \
--defsym ___rombase=0xA0000000 --defsym ___romsize=0x00800000 \
%(start_and_stack)

*board_link:
%(mb411)
```

This technique allows more than one configuration to be defined (although in this example the `board_link` spec string still has to be manually altered in order to switch between them).

An example of using the `-mboard` option is as follows:

```
*start_and_stack:
--defsym _start=___rambase+.reservedramsize \
--defsym _stack=___rambase+___ramsize-4


*mb411:
--defsym .reservedramsize=0x1000 \
--defsym ___rambase=0xA4000000 --defsym ___ramsize=0x02000000 \
--defsym ___rombase=0xA0000000 --defsym ___romsize=0x00800000 \
%(start_and_stack)


*mb442:
--defsym .reservedramsize=0x1000 \
--defsym ___rambase=0xA4000000 --defsym ___ramsize=0x02000000 \
--defsym ___rombase=0xA0000000 --defsym ___romsize=0x00800000 \
%(start_and_stack)


*board_link:
%{mboard=mb411|!mboard=*:%(mb411); \
  mboard=mb442:%(mb442)}
```

This defines that “if the option `-mboard=mb411` was specified, or if no `-mboard` option was specified then use the spec string `mb411`” (and is therefore the default action). The next line defines exactly the same but for `mb442` and both these entries are exclusive. This technique can be scaled up to any number of boards.

An example of adding memory region support is as follows:

```
*start_and_stack:
--defsym _start=___rambase+.reservedramsize --defsym _stack=___rambase+___ramsize-4

*region_p0:
--defsym .regionbase=0

*region_p1:
--defsym .regionbase=0x80000000

*region_p2:
--defsym .regionbase=0xA0000000

*region_p3:
--defsym .regionbase=0xC0000000

*define_29bit_mem:
--defsym .reservedramsize=0x1000 \
--defsym ___rambase=.regionbase+.physrambase \
--defsym ___rombase=.regionbase+.physrombase \
%(start_and_stack)

*_mb411:
--defsym .reservedramsize=0x1000 \
--defsym .physrambase=0x04000000 --defsym ___ramsize=0x02000000 \
--defsym .physrombase=0x00000000 --defsym ___romsize=0x00800000

*mb411:
%(_mb411) %(region_p0) %(define_29bit_mem)
```

```
*mb411p1:
%(_mb411) %(region_p1) %(define_29bit_mem)

*mb411p2:
%(_mb411) %(region_p2) %(define_29bit_mem)

*mb411p3:
%(_mb411) %(region_p3) %(define_29bit_mem)

*board_link:
%{mboard=mb411:%(mb411p1);\
  mboard=mb411p0:%(mb411p0);\
  mboard=mb411p1:%(mb411p1);\
  mboard=mb411p2:%(mb411p2);\
  mboard=mb411p3:%(mb411p3)}
```

This is similar to adding new boards except that instead of specifying a default board, a default memory region (P1) is specified.

When linking for a multicore SoC with one or more ST200 cores, the top 16 Kbytes of memory is normally reserved for the ST200 core debug. When this overlaps RAM that is otherwise allocated to an ST40, this space must be reserved to prevent it being overwritten by the ST40. In the `boardspecs` file, this is done using the symbol `.st200debugramsize`.

The following example is for the `mb411lmivid` variant, and reserves 16 Kbytes (`0x4000` bytes) of memory to prevent the ST40 using this space for its own stack.

```
*_mb411lmivid:
--defsym .reservedramsize=0x1000 --defsym .st200debugramsize=0x4000 \
--defsym .physrambase=0x10000000 --defsym ___ramsize=0x04000000-.st200debugramsize \
--defsym .physrombase=0x00000000 --defsym ___romsize=0x00800000
```

Adding support for new boards

See *Section 9.2.1: Creating a customized board support library on page 130* and *Section 9.3: Adding support for new boards on page 131* for information on how to add and use a new board support library with the toolset.

3.5.2 Linker board support

The board support mechanism (available only in the SuperH configuration) permits the location the executable is placed in memory and the location of its stack to be specified on the command line by defining the symbols `_start` and `_stack`. For example:

```
sh4ld --defsym _start=0xA4001000 --defsym _stack=0xA5FFFFFC ...
```

The start address (`_start`) should be near the beginning of memory. By default, a small area (4 Kbytes) is reserved before the `_start` symbol. This is done using the `.reservedramsize` symbol. This is not required by the ST40 Micro Toolset, but is consistent with other ST40 targeted GNU based toolsets, which use this value for historical reasons. For examples, see *Section 3.5.1 on page 42*.

The stack address (`_stack`) specifies the initial location of the stack for the application and is the location at the end of the memory allocated to the executable (the ST40 uses a falling stack).

Note: *If the stack is to be at the top of memory, the address should be one word less than the top of memory since the location must be a legal address.*

The GCC compiler driver passes these options automatically when it is used to perform the link.

The default linker script enables the linking of run-from-ROM executables. `_start` defines the location of the start of ROM for the read-only sections, and `.start_rwdata` the location in RAM where the read-write sections begin. When a standard `-mboard` option is specified, the symbols `___rambase`, `___ramsize`, `___rombase` and `___romsize` can be used to define the values of `_start` and `.start_rwdata`, for example:

```
sh4gcc -Wl,--defsym,_start=___rombase+0x5000,--defsym,
       .start_rwdata=__rambase
```

3.5.3 Alternative placement of sections

The default linker script expects all code and data sections to be contiguous in the `.text`, `.rodata` and `.data` sections. For some applications, it may be necessary for code and data to be located in a different memory location to the rest of the application.

For example, in a low-power scenario, the RAM on the LMI may be put into self-refresh mode (and thus made inaccessible to the ST40). The code and data required to do this needs to be located in RAM that is not connected to the LMI. This can be achieved either by using of a relocatable library loaded into that location, or by creating alternative linker sections in the main application at the required addresses.

Note: OS21 provides power management APIs that achieve the same objective but do not require the code and data to be placed in alternative sections.

The `os21/rombootanywhere` example demonstrates the creation and placement of alternative sections by replacing the default linker script with one that includes the default linker script and then lays out the alternative sections. The example uses the GCC `section` attribute in the source code to indicate to the script where the code and data should be located.

3.6 Run-time support

The run-time support mechanism allows GCC to link programs for each recognized run-time system. The run-time is specified using the `-mruntime` option (see *Section 3.6.1: GCC run-time support setup*).

Table 12 lists the recognized run-time systems.

Table 12. Recognized run-time systems

Supported run-time systems	Comment
`bare`	Bare machine (default)
`os21`	OS21
`os21_d`	OS21 debug

3.6.1 GCC run-time support setup

The GCC -mruntime option is controlled by the GCC specs file runtimespecs. This file is located in the subdirectory sh-superh-elf/lib/gccscripts of the release installation directory. A description of the format of a GCC specs file may be found in *Section 3.15* of the *Using the GNU Compiler Collection* manual.

The SuperH configuration adds five spec strings in the standard GCC specs file. (These can be reviewed using the GCC option -dumpspecs.) They are named asruntime (assembler), cppruntime (C preprocessor), cc1runtime (compiler), ldruntime (linker) and libruntime (libraries). These spec strings can be overridden in the runtimespecs file in order to specify a new run-time environment setup.

An example of the simplest runtimespecs file is:

```
*asruntime:
               (line intentionally blank)

*cppruntime:
-D__BARE_BOARD__

*cc1runtime:
               (line intentionally blank)

*ldruntime:
               (line intentionally blank)

*libruntime:
-lc -ldtf
```

This sets the run-time environment to that of a bare machine application (that is, an application without an operating system) using the Data Transfer Format (DTF) I/O library. It does not provide information about which board is targeted. There is one entry for each of the five spec strings.

Each of the five spec strings is inserted respectively into the command lines of the assembler, preprocessor, compiler and linker (general linker options and library options).

There is an implicit -lc placed at the end of the library spec string libruntime. However, if a file or library listed in the libruntime string provides a symbol required by the C library (such as those found in **libgloss** or **libdtf**) then it is necessary to place an additional -lc first. The final line of the example shows -lc is listed before -ldtf.

The previous example allows only one run-time to be defined and ignores the -mruntime option. A simple example supporting both bare machine and OS21 applications is:

```
*as_bare:
               (line intentionally blank)

*cpp_bare:
-D__BARE_BOARD__
```

```
*cc1_bare:
               (line intentionally blank)

*ld_bare:
               (line intentionally blank)

*lib_bare:
-lc -ldtf

*libgcc_bare:
-lgcc

*as_os21:
               (line intentionally blank)

*cpp_os21:
-D__os21__ -D__OS21_BOARD__

*cc1_os21:
               (line intentionally blank)

*ld_os21:
               (line intentionally blank)

*lib_os21:
%(lib_bare) -los21 -l%(lib_os21bsp_base) -los21 %(lib_bare)

*libgcc_os21:
-los21 %(libgcc_bare)

*asruntime:
%{mruntime=bare|!mruntime=*:%(as_bare);\
  mruntime=os21:%(as_os21)}

*cppruntime:
%{mruntime=bare|!mruntime=*:%(cpp_bare);\
  mruntime=os21:%(cpp_os21)}

*cc1runtime:
%{mruntime=bare|!mruntime=*:%(cc1_bare);\
  mruntime=os21:%(cc1_os21)}

*ldruntime:
%{mruntime=bare|!mruntime=*:%(ld_bare);\
  mruntime=os21:%(ld_os21)}
```

```
*libruntime:
%{mruntime=bare|!mruntime=*:%(lib_bare);\
  mruntime=os21:%(lib_os21)}

*libgcc:
%{mruntime=bare|!mruntime=*:%(libgcc_bare);\
  mruntime=os21:%(libgcc_os21)}
```

Note: *Refer to the installed* `boardspecs` *and* `runtimespecs` *GCC specs files for the definition of the* `lib_os21bsp_base` *spec string.*

Again, there is one line to describe which spec string is being defined (or redefined), one (possibly blank) line defining the spec string (after escape processing) and one blank line between rules.

The example supports the compiler driver options:

```
-mruntime=bare
-mruntime=os21
```

4 Cross development tools

The cross development tools enable an executable that has been created by the code development tools (see *Chapter 3: Code development tools on page 34*), to run on a variety of simulator and hardware platforms through the GNU debugger (GDB).

GDB has been enhanced in the SuperH configuration (see *Section 1.2: The SuperH configuration on page 16*) to provide better support for the ST40 simulator and silicon targets.

For the ST40, there are two methods to configure a target through an ST Micro Connect.[a]

- ST TargetPacks – for targets connected to any type of STMC.
- GDB command scripts – for targets connected to an STMC1 or STMCLite. GDB command scripts are also the only method for connecting and configuring simulated targets. For more information about using GDB command scripts, see *ST40 Micro Toolset GDB command scripts user manual* (8045872)

Note: 1 *For backwards compatibility, a target connected to an STMC2 may still be configured using a GDB command script. The command script must, however, be modified to make the connection using the appropriate ST TargetPack. See Using a GDB script with an STMC2 on page 51 for more information.*

2 *The STMC software and ST TargetPacks, together known as the ST Micro Connection Package, are available as a separate release to the ST40 Micro Toolset. STMicroelectronics recommends installing the most up-to-date version of the ST Micro Connection Package. For information concerning the ST Micro Connection Package available for the ST Micro Connect 2, contact your ST FAE or ST support center.*

4.1 Connecting to the target

4.1.1 Using an ST TargetPack

A connection from a host to a target through an ST Micro Connect can be made either by running **sh4xrun** or directly using GDB. The purpose of an ST TargetPack is to provide a description of the target, in order for the ST Micro Connect to configure the target.

The connection is made by using a target connection command (such as `sh4tp`) and an appropriate TargetString with **sh4xrun** or GDB.

The TargetString is made up of three elements, delimited by colons, as follows:

```
name:targetpack:core
```

where *`name`* is the IP address or name of the ST Micro Connect, *`targetpack`* is the name of the ST TargetPack to use to configure the target, and *`core`* is the name of the core that is defined by the ST TargetPack (for an SoC that contains a single ST40 core, this is usually

[a] The original ST Micro Connect product was named the **ST Micro Connect**. With the introduction of ST Micro Connect 2 and the ST Micro Connect Lite, this product is now known as ST Micro Connect 1 and the generic term **ST Micro Connect** refers to the family of ST Micro Connect devices. In some instances, the names are abbreviated to STMC, STMC1, STMC2 and STMCLite.

st40). A list of current SoCs and the names of the cores to which a connection can be made is given in the release notes to the ST Micro Connection Package.

The following example uses **sh4xrun** (see *Section 4.3: Using sh4xrun*) to execute the application a.out on an STb7100-MBoard connected to the STMC called stmc using an ST TargetPack.

```
sh4xrun -c sh4tp -t stmc:mb411:st40 -e a.out
```

This example calls the sh4tp command for the TargetString given by the -t option and runs the executable a.out.

To execute the same application using GDB:

```
sh4gdb -ex “sh4tp stmc:mb411:st40“ -ex load -ex continue a.out
```

In this case, the sh4tp command and TargetString are passed directly to the debugger as a quoted argument using the GDB -ex command line option.

The sh4tp command can also be used within GDB:

```
(gdb) sh4tp stmc:mb411:st40
```

ST TargetPacks are available separately from the ST40 Micro Toolset. Contact your ST FAE or ST support center to obtain the latest version of the ST TargetPack for the ST hardware platform that you intend to use.

4.1.2 Using a GDB script with an STMC1 or STMCLite

If the STb7100-MBoard is connected to an ST Micro Connect 1 or ST Micro Connect Lite (called stmc in this example), then to use a GDB command script instead of an ST TargetPack, use the following command:

```
sh4xrun -c mb411bypass -t stmc -e a.out
```

where mb411bypass is the command script for connecting to the target.

See *ST40 Micro Toolset GDB command scripts* (8045872) for information on using GDB command scripts.

4.1.3 Using a GDB script with an STMC2

If the target is connected to an ST Micro Connect 2, the same command script may be used, but it must be modified to use the appropriate ST TargetPack for the target. The following script is an example of a connection command for the STb7100-MBoard.

```
define mb411stb7100
   source register40.cmd
   source display40.cmd
   source stb7100clocks.cmd
   source stb7100jtag.cmd
   source stb7100.cmd
   source mb411stb7100.cmd
   source sh4connect.cmd
   source plugins.cmd
   if ($argc > 1)
      connectsh4le $arg0 mb411stb7100_setup $arg1
   else
      connectsh4le $arg0 mb411stb7100_setup "jtagpinout=st40 hardreset"
   end
end
```

To convert this GDB script for the STMC2, make the following modifications:

- change the call to `connectsh4le` to `sh4tp` with the name of the appropriate TargetPack as a parameter
- make a call to the board-specific function to setup the target (`mb411stb7100_setup` in this case)

The call to `sh4tp` must appear exactly as follows:

```
sh4tp $arg0:mb411:st40,no_pokes=1,no_devid_validate=1,tck_frequency=3375000
```

Caution: If the command is not entered exactly as written above, the connection to the target may fail. Do not alter the TargetString comma separated parameters.

The STMC2 version of the script is:

```
define mb411stb7100stmc2
   source register40.cmd
   source display40.cmd
   source stb7100clocks.cmd
   source stb7100jtag.cmd
   source stb7100.cmd
   source mb411stb7100.cmd
   source sh4connect.cmd
   source plugins.cmd

   sh4tp $arg0:mb411:st40,no_pokes=1,no_devid_validate=1,tck_frequency=3375000

   mb411stb7100_setup
end
```

This command script expects that the name of the ST Micro Connect is passed as a command line argument (`$arg0`) and uses the `mb411` ST TargetPack to perform the initial connection to the target. The additional `sh4tp` parameters prevent the ST TargetPack from performing any initialization of the peripherals. Instead, this is performed by the command script `mb411stb7100_setup`.

Information on the parameters passed to the connection command with the TargetString can be found in the *Developing with an ST Micro Connect and ST TargetPacks application note* (8174498).

See *ST40 Micro Toolset GDB command scripts* (8045872) for information on using GDB command scripts to connect to targets.

4.1.4 Auto-detect connection

The connection commands have the ability to detect the type of connection being used and set up the hardware accordingly. The auto-connect mechanism uses the name of the ST Micro Connect to ascertain the type of connection to use. *Table 13* lists the names that are always assumed to specify USB connections for the given STMC type.

Table 13. STMC names assumed to specify USB connections

Name	STMC type	Host
`STMCLT`*`digit+`*	STMCLite	Linux, Windows
`USB`	STMC1 USB	Windows[1]
`HTI`*`digit`*	STMC1 USB	Windows[1]

[1] The check of the STMC name is case insensitive.

There may be certain circumstances under which the STMC might fail to identify the connection correctly. In order to overcome this problem, it is possible to override the auto-detection mechanism by replacing the STMC name in the connection command or in the TargetString with one of the following strings:

- to force the use of an STMC1 Ethernet connection to *`stmc`*:
  `stmc1@eth=`*`stmc`*
- to force the use of an STMC1 USB connection to *`stmc`*:
  `stmc1@usb=`*`stmc`*
- to force the use of an STMC2 Ethernet connection to *`stmc`*:
  `stmc2=`*`stmc`*
- to force the use of an STMCLite USB connection to *`stmc`*:
  `server@stmclite=`*`stmc`*

For example, to force the host to use an STMC1 Ethernet connection to connect to an MB519-STi7200 target through an STMC1 with the name `hti1`, use either the following GDB script or ST TargetPack connection command:

- GDB script:
  `mb519bypass stmc1@eth=hti1`
- ST TargetPack:
  `sh4tp stmc1@eth=hti1:mb519:st40`

4.1.5 Identification of the STMCLite

When connecting to a target attached to an STMCLite, the STMCLite serial number forms part of the ST Micro Connect name that is passed to the connection command. The full name consists of the following components:

- STMCLite serial number
- Target interface identifier, which is **A** for the Target1 connector and **B** for the I^{2}C/Target2 connector. See the *ST Micro Connect Lite Datasheet* (8282486) for information on the different connectors.

On Windows, the name used to connect to a target attached to an STMCLite is constructed by concatenating the serial number and interface identifier.

On Linux, the serial number and interface identifier are separated by a space. However, for convenience, the ST40 tools allow an underscore (_) or a hyphen (-) to be used instead of a space[b].

Examples

Assuming an STMCLite with the serial number of `STMCLT1000` is attached to an STi7108-HDK reference board with the Target1 connector, the following GDB commands connect to the host ST40 CPU of the target:

- Windows:

  ```
  sh4tp STMCLT1000A:hdk7108stx7108:host
  ```
- Linux:

  ```
  sh4tp STMCLT1000_A:hdk7108stx7108:host
  ```

Note: The name (for example, `STMCLT1000A`*) is case sensitive and therefore must be specified in upper case.*

4.2 The GNU debugger

The GNU debugger (GDB) supports the downloading and debugging of applications on:

- silicon (using an ST Micro Connect)
- the ST40 functional simulator
- the ST40 performance simulator
- the GDB built-in simulator

The following interfaces are available for debugging applications using GDB:

- the STWorkbench IDE (see *Chapter 5: Using STWorkbench on page 68*)
- **sh4gdb**, which uses the standard command line interface
- **sh4insight**, which uses the Insight GUI (described in *Chapter 6: Using Insight on page 75*)

sh4insight is identical to **sh4gdb** except that it defaults to starting the Insight GUI instead of the command line interface. Therefore, wherever **sh4gdb** is referenced the same also applies to **sh4insight**.

This section describes only the standard command line interface, **sh4gdb**.

4.2.1 Using GDB

GDB can be used to execute any program, but it can only be used effectively to debug programs compiled with debugging information (using the `-g` compilation option).

When the program is compiled, start GDB as follows:

```
sh4gdb executable
```

GDB shows a message describing its version and configuration followed by a command prompt `(gdb)`.

[b] In a future version of the STMCLite Linux driver, the difference between the Windows and Linux naming conventions may disappear and only the Windows convention will apply.

There are many GDB commands available. For full instructions on all these commands use the GDB `help` command or refer to the GNU *Debugging with GDB* manual.

All of the commands may be abbreviated to the shortest name that is still unique. The GDB command line supports auto-completion of both commands and, where possible, parameters. In addition, many of the most common commands have single letter aliases.

Most of the commands serve no purpose until GDB has connected to and initialized a target.

Connecting to a target

There is a range of different target types that can be used to debug the executable. The connection command varies according to the target type.

- Connect to the simulator built into GDB with the `target sim` command.
- Connect to the ST40 simulators and silicon (using an ST Micro Connect) with special GDB commands (see *ST40 Toolset GDB command scripts* (8045872) or by using ST TargetPacks (see *Section 4.1: Connecting to the target on page 50*).

  The commands are only available when using the short form of GDB (**sh4gdb**). For example, to connect to a standard ST40 simulator configured for the STb7100-MBoard:

  ```
  (gdb) mb411sim
  ```

  When using an ST TargetPack, use the `sh4tp` command. The following is an example of an `sh4tp` command:

  ```
  (gdb) sh4tp stmc:mb411:st40
  ```

A connection can also be made from the command line, when first invoking the debugger. The following example executes an application called `a.out` using an STMC called `stmc` on an STb7100-MBoard.

```
sh4gdb -ex “sh4tp stmc:mb411:st40“ -ex load -ex continue a.out
```

In this case, the `sh4tp` command and its arguments (the TargetString) are passed directly to the debugger as a quoted string using the GDB `-ex` command line option.

Executing the program

If the program is executed immediately, it simply runs until it reaches completion or an error. In many cases it is desirable to set a breakpoint so that the program stops at the point of interest and allows inspection or single-stepping of the program state.

Breakpoints can be set on specific functions, lines or addresses using the `break` command, for example:

```
(gdb) break main
(gdb) break 42
(gdb) break *0x08002000
```

When ready, download and start the program on the target by invoking the `run` command.

Note: *It is possible to specify arguments to the* `run` *command. These arguments are passed to the target program, which are accessible through* `argc` *and* `argv` *as usual.*

For the GDB simulator target, the `load` command must be used to download the program onto the target before the `run` command is used. For other targets, only the `run` command is required. However, in all cases `continue` must be used to restart the program after it has stopped.

The program runs until it completes, hits a breakpoint, is interrupted by the user with a **Ctrl+C**, or encounters an error. At this point, a short explanatory message is displayed and the GDB prompt returns.

The following commands are provided to step execution a line, or a machine instruction at a time:

- the `step` command (abbreviated to `s`) moves on to the next source line (even if it is in a different function)
- the `stepi` command (abbreviated to `si`) moves on a single machine instruction before pausing the program again
- the `next` command (abbreviated to `n`) is the same as `step`, but moves to the next line in the current function, rather than the next line in the program, stepping over any function calls
- the `nexti` command is the machine code equivalent of `next`, it moves to the next instruction in the sequence even if the current one is a call

Examining the target

All the GDB commands for interrogating targets are available.

To view the register set, use the `info registers` and `info all-registers` commands.

To disassemble the current function, use the `disassemble` command.

To disassemble the current instruction, use:

```
(gdb) x/i $pc
```

To inspect the memory, use the `x` (examine) command. For example:

```
(gdb) x 0x1000
```

For other formats, use the `/` modifier. For example, to display memory as strings:

```
(gdb) x/s 0x08001234
```

To view by name any variable currently in scope, use the `print` (or `p`) command. The command can also be used with expressions, for example:

```
(gdb) p foo+bar*2
```

To format the displayed information, use the `printf` command. For example:

```
(gdb) printf "%s %d %d\n", 0x8001234, foo, foo+bar*2
```

Changing the state of the program

To alter memory locations, registers and variables, use the `set` command. For example:

```
(gdb) set variable i = 0
```

The expression syntax is much the same as C (or C++ depending what is being debugged), but there are some extensions. Refer to the GNU *Debugging with GDB* manual for details.

Exiting GDB

When the debug session is complete, exit GDB using the `quit` (or `q`) command.

4.2.2 The .shgdbinit file

On startup, GDB searches for a file named `.shgdbinit`, first in the home directory and then in the current working directory. If either or both of these files exist, GDB sources their contents.

The GDB `-nx` option prevents GDB from sourcing any of these files.

Note: Any commands in the `.shgdbinit` *files that require confirmation assume affirmative responses. Any line beginning with* `#` *will be ignored.*

Additionally, if GDB is launched using the **sh4gdb** or **sh4insight** tools, a default `.shgdbinit` file is sourced before any other file (see *Using the sh-superh-elf-gdb or sh-superh-elf-insight tools on page 57*). This file enables support for the ST40 simulator and silicon targets. The `-nx` option has no effect on sourcing this file.

Using the sh4gdb or sh4insight tools

When using the **sh4gdb** or **sh4insight** tools to launch GDB, there is no requirement to create any additional `.shgdbinit` files. However, the `.shgdbinit` files are still useful for setting up user preferences and defaults.

Using the sh-superh-elf-gdb or sh-superh-elf-insight tools

When using the **sh-superh-elf-gdb** or **sh-superh-elf-insight** tools to launch GDB, a user-defined `.shgdbinit` file can enable the ST40 simulator and silicon support. Use the `source` command to source the default `.shgdbinit` (in the `sh-superh-elf/stdcmd` subdirectory of the release installation directory).

The default `.shgdbinit` file assumes that the `sh-superh-elf/stdcmd` directory is on the GDB search path, which can be displayed using the GDB `show directories` command and set using the GDB `dir` command.

The standard command scripts, containing the SuperH configuration and target connection mechanisms, are also located in the `sh-superh-elf/stdcmd` directory and can be identified by the `.cmd` extension.

4.2.3 Connecting to a running target

The ST40 Micro Toolset supports connecting to a running target. This feature allows GDB to gain control of the target without disrupting the application already running on the target. A typical situation where this would be used is when an application has booted from Flash ROM.

The ST40 Micro Toolset provides similar support for connecting to a target that is stopped in *debug* mode (see below).

Using an ST TargetPack

The command to connect to a running target, connected to the STMC called `stmc` (assuming that the target is an STb7100-MBoard) is the following:

```
(gdb) sh4tp stmc:mb411:st40,no_reset=1,no_pokes=1
```
[c]

The `no_reset=1` and `no_pokes=1` TargetString parameters indicate that a connection to a running target is requested. These parameters prevent the ST TargetPack from resetting and re-configuring the target, which would destroy its state.

The command to reconnect to the same target stopped in debug mode is:

```
(gdb) sh4tp stmc:mb411:st40,no_reset=1,no_pokes=1 resettype=none
```

Note: A target is stopped in debug mode if it has not been reset since it was last connected to by GDB and was disconnected from GDB with the `ondisconnect=none` *configuration option set (the default behavior).*

For details about the `resettype=none` and `ondisconnect=none` configuration options, see the *ST40 Toolset GDB command scripts* (8045872).

Using a GDB script

The ST40 Micro Toolset also provides specialized *attach* connection commands for targets connected to an STMC1 or STMCLite.

For a description of these connection commands for attaching to a running target and to a stopped target, see the *ST40 Toolset GDB command scripts* (8045872).

[c] If using version R1.1.1 or earlier of the ST Micro Connection Package then add the option `resettype=break` to the command for connecting to a running target. This option is not required when using later versions of the ST Micro Connection Package.

4.2.4 GDB command line reference

Table 14 lists some of the most useful command line options.

Table 14. sh4gdb command line options

Option	Description
`-nw` `-nowindows`	Disable the Insight GUI and use the command line interface. Equivalent to the option `-interpreter=console`.
`-n` `-nx`	Prevent GDB from sourcing any `.shgdbinit` files or reading the `.gdbtkinit` file (if they exist). If the environment variable `INSIGHT_FORCE_READ_PREFERENCES` is set, then `-nx` does not prevent the reading of the `.gdbtkinit` file.
`-w` `-windows`	Enable the Insight GUI instead of the command line interface, see *Chapter 6: Using Insight on page 75*. Equivalent to the option `-interpreter=insight`.
`-tui`	Enable the GDB text user interface (TUI) instead of the command line interface. Equivalent to the option `-interpreter=tui`.
`-args` *`exe args`*	Debug the program (*`exe`*) and pass the command line arguments *`args`* to the program (*`exe`*).
`-batch`	Process the command line options (including any scripts from the `-command` option) and then exit.
`-batch-silent`	This option is similar to `-batch` except that the debugger suppresses all messages other than errors.

`-command` *`file`*
`-x` *`file`*
	Source the commands in *`file`*. This is useful for setting up functions or automating downloads.

`-eval-command` *`command`*
`-ex` *`command`*
	Execute the specified GDB command, *`command`*. This option may be specified multiple times to execute multiple commands. When used in conjunction with `-command`, the commands and scripts will be executed in the order specified on the command line.

`-interpreter` *`interface`*
`-ui` *`interface`*
`-i` *`interface`*
	Set the GDB user interface to *`interface`*. Standard user interfaces are `console`, `tui`, `insight` and `mi`.

`-return-child-result`	The return value given by GDB will be the return value from the target application (unless an explicit value is given to the GDB `quit` command, or an error occurs).

4.2.5 GDB command quick reference

Table 15 lists some of the most useful GDB commands. It does not include any of the additional commands for connecting and controlling targets that have been added in the SuperH configuration. For details of these see *Section 4.2.6: Additional GDB commands on page 62*. The *Debugging with GDB* manual provides further details on GDB commands and the GNU debugger.

Table 15. sh4gdb command quick reference

Command	Description
`$argc`	A GDB convenience variable automatically defined at the start of every user defined GDB command that specifies the number of arguments to the command. This allows a command to test how many parameters have been passed to it.
`backtrace` *`n`* `[full]`	Print a backtrace of all the stack frames (function calls). If *`n`* is specified and is positive then give the top *`n`* frames. If *`n`* is specified and is negative then give the bottom *`n`* frames. If the word `full` is given then it also prints the values of the local variables. The `bt` command may be used as an alias for `backtrace`.
`break` *`function`*|*`line`* |*`file:line`* |*`*address`*	Set a breakpoint on the specified function, line or address.
`clear` *`function`*|*`line`* |*`file:line`* |*`*address`*	Clear a breakpoint on the specified function, line or address.

`compare-sections`
`[`*`section-name`*`]`
[*LMA* | *VMA*] [*`offset`*]
	Compare the data of the loadable section *`section-name`* in the executable file of the program being debugged with the same section in the remote machine's memory, and report any mismatches. With no arguments, compares all loadable sections.
	`LMA` (Load Memory Address) or `VMA` (Virtual Memory Address) specify whether GDB compares the LMA or the VMA addresses from the executable file. If unspecified, `LMA` is assumed.
	`offset` specifies the offset to add to the address of each section loaded into memory. The default is `0`.

`continue`	Continue execution of the program.
`delete [`*`number`*`]`	Delete the numbered breakpoint or all breakpoints.
`disable [`*`number`*`]`	Disable the numbered breakpoint or all breakpoints.

`disassemble [`*`add1`*`],`
`[`*`add2`*`]`
	Disassemble the machine code between the addresses *`add1`* and *`add2`*. If one address is omitted then the code around the one given is disassembled. If both are omitted then it uses the program counter as the address to use. (Command syntax changed for GDB 7.1.)

`enable [`*`number`*`]`	Enable the numbered breakpoint or all breakpoints.
`file` *`file`*	Use *`file`* as the program to be debugged.
`finish`	Complete current function.
`help`	GDB commands assistance.
`info all-registers`	Print the contents of all the registers.
`info breakpoints`	List all breakpoints.

Table 15. sh4gdb command quick reference (continued)

Command	Description
`info registers`	Print the contents of the registers.
`init-if-undefined` *`var = exp`*	The same as the GDB `set variable` command except that it does not overwrite an existing value. *`var`* must be a GDB convenience variable.
`list`	List next ten source lines.
`list -`	List previous ten source lines.
`list` *`function`*|*`line`* |*`file:line`* |*`*address`* |*`file:function`*	List specific source code. Any two arguments separated by a comma are required to specify a range.
`load [`*`file`*`]` [*LMA* | *VMA*] [*`offset`*]	Download the `file` to the target. If no *`file`* is given, the executable from the GDB command line or the `file` (or `exec-file`) command is used. `LMA` (Load Memory Address) or `VMA` (Virtual Memory Address) specify the area of memory *`file`* is copied to. If unspecified, `LMA` is assumed. `offset` specifies the offset to add to the address of each section loaded into memory. The default is `0`.
`next [`*`n`*`]`	Continue execution to next source line, stepping over functions. If *`n`* is specified, do this *`n`* times.
`nexti [`*`n`*`]`	Execute exactly one instruction, stepping over subroutine calls. If *`n`* is specified, do this *`n`* times.
`print` *`exp`*|*`$r`*	Print the value of the expression *`exp`* or contents of the register *`$r`* (for example, `$r0` or `$pc`).
`printf "`*`format`*`",` *`arg1, ..., argn`*	Same as `print` but with a format-string. Allows more than one parameter to be printed. Parameters must be separated by commas.
`quit [`*`code`*`]`	Exit GDB with the return value *`code`*, if specified. If *`code`* is not specified, GDB will exit with the return value of 0. Note that the GDB convenience variable `$_exitcode` is set to the return value of the target application and therefore may be used as the value for *`code`*, for example `quit $_exitcode`.
`rbreak` *`regexp`*	Set a breakpoint on all functions that match *`regexp`*.
`run [`*`file`*`]` *`args`*	Run the program. The program must already have been downloaded (using `load`) when using the GDB simulator. If an executable was given on the command line then *`file`* must not be given here.
`set args` *`args`*	Set the command line for the program. Used before starting the program.
`set variable` *`var = exp`*	Set the value of a variable or register.
`step [`*`n`*`]`	Continue execution to next source line. If *`n`* is specified, do this *`n`* times.
`stepi [`*`n`*`]`	Execute exactly one instruction. If *`n`* is specified, do this *`n`* times.
`set trace-commands on|off`	Set trace-commands `on`/`off`. Enables the tracing of GDB commands. The default is `off`.
`show trace-commands`	Displays the current state of GDB CLI command tracing.

Table 15. sh4gdb command quick reference (continued)

Command	Description
`target sim`	Use the GDB built-in simulator instead of the ST40 simulator or silicon target.

`tbreak` *`function`*`|`*`line`*
`|`*`file:line`*
`|`*`*address`*
	Set a temporary (one time only) breakpoint on the specified function, line or address.

`watch` *`exp`*	Set a watchpoint for the expression *`exp`*.
`where` *`n`* `[full]`	This is identical to the `backtrace` command.

4.2.6 Additional GDB commands

There are several additional features in the supplied GDB not found in the standard version from the Free Software Foundation (FSF). Some of these features are not specific to the SuperH configuration, but are generic features that have been added in order to provide better support for the implementation of the GDB scripts used for connecting to ST40 simulator and silicon targets.

Table 16 lists the additional GDB commands not specific to the SuperH configuration.

Table 16. Additional sh4gdb commands (not SuperH specific)

Command	Description
`${`*`variable`*`}`	Substitute `${`*`variable`*`}` with the contents of the environment variable called *`variable`*. Note that substitutions are recursive; that is, if `${`*`var1`*`}` is replaced with a value that contains `${`*`var2`*`}`, then `${`*`var2`*`}` is also replaced with its value.

`fork` *`command arg1`*
`[`*`arg2 ... argn`*`]`
	Execute the host command *`command`* with the specified arguments (*`arg1`* to *`argn`*).
	The host starts *`command`* with the following command line:
	`command fdout fdin arg1 arg2 ... argn`
	where:
	`fdout` is the file descriptor of a writable pipe. Anything written to this pipe is interpreted as commands by GDB.
	`fdin` is the file descriptor of the read end of the same pipe.

`keep-variable` *`var`*	This command is provided for backwards compatibility. GDB always keeps convenience variables.

`set backtrace`
`abi-sniffer`
`on|off`
	Use the ABI frame analyzer for back tracing (in addition to the DWARF2 debug information). Use this to obtain back traces when debug information is not available. The default is `on`.

`set silent-commands`
`on|off`
	Set the output state for GDB commands. If set to `on` then no output is displayed by GDB commands. The default is `off`.

`show backtrace`
`abi-sniffer`
	Display the current state of the ABI frame analyzer.

`show silent-commands`	Display the output state for GDB commands.
`sleep` *`time1`* `[`*`time2`*`]`	Sleep for the specified time. *`time1`* is given in seconds and *`time2`* (optional) is given in microseconds.

Table 17 lists the additional GDB commands that are specific to the SuperH configuration.

Table 17. SuperH configuration specific sh4gdb commands

Command	Description
`console on|off`	Synonym for `enable console` and `disable console`.
`disable console | rtos | sharedlibrary`	Disable the feature represented by the keyword `console`, `rtos` or `sharedlibrary`. All three are enabled by default. For information about `disable console`, see *Section 4.2.7: Console settings on page 65*. For information about `disable rtos`, see *Section A.7.3: Debugging OS21 boot from ROM applications on page 237*. `disable sharedlibrary` disables the debug support for shared libraries. When this facility is disabled, GDB does not examine memory or insert breakpoints to monitor the loading and unloading of shared libraries.
`enable console | rtos | sharedlibrary`	Enable the feature represented by the keyword `console`, `rtos` or `sharedlibrary`. All three are enabled by default. See `disable` for more information.
`maintenance shtdi rtos-reset all | thread`	Reset the cached RTOS state.
`msglevel` *`opt`*	Set the target debug interface message level. Use `help msglevel` for a list of valid options.
`rtos on|off`	Synonym for `enable rtos` and `disable rtos`.
`set shtdi` *`subcommand [option]`*	Configure the SHTDI interface by applying the given *`subcommand`* and *`option`*. Details of the subcommands and their permitted options are given in *Table 18*.
`show shtdi` *`subcommand`*	Show the current value of the corresponding `set shtdi` *`subcommand`*. Details of the subcommands are given in *Table 18*.
`target shtdi`	Use the SuperH target debug interface to connect to a target. Used by the GDB connection commands.

The subcommands accepted by `set shtdi` and `show shtdi` are listed in *Table 18*.

Table 18. Subcommands available with the set shtdi and show shtdi commands

Subcommand	Description
`break-timeout` *`n`*	Set the timeout to wait before breaking into the target. The timeout *`n`* is specified in seconds. Set to 0 to disable. The default is 0.
`continue-after-exit on | off`	Allow GDB to continue debugging the target after the application calls `exit()`. The default is `off`.
`read-all-registers on | off`	Enable or disable the ability to read all CPU registers as a single request, instead of reading each register individually. Enabling this operation may produce an appreciable improvement in performance. The default is `on`.
`rtos-initialize-hook` *`option`*	Configure when RTOS awareness is to be enabled. *`option`* is one of the following: – `attach`: enable after attaching to the target – `load`: enable after loading an application – `resume`: enable after starting the application The default is `resume`. Use this command before connecting to a target.
`rtos-thread-information on | off`	Enable or disable the display of task status information by the `info threads` command. Disabling this operation may produce an appreciable improvement in performance. The default is `on`. STWorkbench, however, sets this command to `off`.
`wait-timeout` *`n`*	Set the timeout when waiting for an event from the target. The timeout *n* is specified in milliseconds. The default is 100 milliseconds.

The `directory` command is usually used by GDB to locate files (such as C source files). It has been extended so that it can also be used to locate GDB command scripts sourced using the `source` command.

There are also a number of commands that are defined when GDB sources the command scripts in the `sh-superh-elf/stdcmd` directory. These command scripts are automatically sourced when **sh4gdb** and **sh4insight** are used. However if **sh4gdb** and **sh4insight** are not used, they must be sourced by a `.shgdbinit` file in order to be available. The commands can be listed using the `help user-defined` command.

4.2.7 Console settings

A target console (separate to the GDB console) is provided for the target application, and is enabled by default. The target console window may be switched off or on at any time using the `disable console` or the `enable console` command respectively.

When `enable console` is specified, the console window is opened and all target I/O is redirected to the new window. When the target program completes, the console window remains open. The console window closes when GDB is quit.

When `disable console` is specified, the console window is closed (if open) and all target I/O is redirected to the same console as GDB. `stdout` and `stderr` are displayed in the GDB command console and `stdin` is read from the GDB command console.

4.3 Using sh4xrun

sh4xrun provides a simple batch mode interface to GDB. This allows users to connect and configure a target system, and to load and execute an application on the target system. **sh4xrun** invokes GDB with all of the options and scripts required to execute the program.

4.3.1 Setting the environment

The setup of **sh4xrun** is identical to the setup of GDB. See *Section 1.3.3: Configuration scripts on page 21*.

4.3.2 sh4xrun command line reference

To display the help, invoke **sh4xrun** with the `-h` option.

Usage

```
sh4xrun {option} [-a | --] {argument}
```

Note: The command order is important; `-a` or `--` must always be the last option as this indicates that all following arguments are to be passed to the target application.

Table 19. sh4xrun command line options

Option	Description
`-c` *`command`*	Specify the target configuration command (GDB command) to be invoked. The configuration command must be compatible with the target. If `-c` is not specified, but the `-t` option is, then an option of `-c sh4tp` is implied by default.
`-d` *`directory`*	Add a directory to GDB’s source search path. This is equivalent to the `-d` GDB command line option. This option can be specified more than once.
`-e` *`filename`*	Specify the executable file to be loaded onto the target.[1]
`-f`	Ignored by **sh4xrun** (included for backward compatibility).
`-g` *`gdbpath`*	Specify the full path to the GDB executable to be used. This should be a version compatible with the version of GDB supplied by STMicroelectronics.

Table 19. sh4xrun command line options (continued)

Option	Description
`-h`	Display the help for **sh4xrun**.
`-i` *`filename`*	Execute the GDB command script *`filename`*. Equivalent to the `-x` GDB command line option. This option can be specified more than once.
`-t` *`target`*	Connect to *`target`*. This can be the target name or IP address (if connecting using a GDB command script) or a TargetString (if connecting using an ST TargetPack). This option is not required for simulator targets.
`-u` *`gdbname`*	Specify the name of GDB. The default is **sh4gdb**.
`-v`	Display verbose information.
`-x` *`filename`*	Execute *`filename`* as the default GDB startup script instead of `.shgdbinit`.
`-A` *`command`*	Execute the GDB command *`command`* after running the program. This option can be specified more than once.
`-B` *`command`*	Execute the GDB command *`command`* before running the program. This option can be specified more than once.
`-C` *`option`*	Add *`option`* to the target configuration command specified by the `-c` option. This option can be specified more than once.
`-D`	Debug (very verbose information).
`-T` *`timeout`*	The maximum permitted time for executing on the target. The command `set shtdi-break-timeout` *`timeout`* is issued to GDB.
`-V`	Display the version of **sh4xrun**.
`-a` `--`	Specify that the remainder of the command line arguments are to be passed as arguments to the target application. This option can only be specified as the final **sh4xrun** specific option in the command line. This option can be omitted if the target arguments do not conflict with the arguments of **sh4xrun**.

[1] If the `-e` option is omitted (and the `-i` option has not been used) then **sh4xrun** assumes that the first argument in the argument list {*`argument`*} is the name of the executable file.

In those cases where an option is specified more than once, the options are applied in the order specified on the command line.

4.3.3 sh4xrun command line examples

To run `hello.out` on an STb7100-MBoard with an ST TargetPack:

```
sh4xrun -t stmc:mb411:st40 -e hello.out
```

The following is also valid:

```
sh4xrun -t stmc:mb411:st40 hello.out
```

Without the `-e` option, **sh4xrun** assumes that the first target argument is the name of the executable.

To run a GDB command before running `hello.out`, use the `-B` option:

```
sh4xrun -t stmc:mb411:st40 -B "stmcprofiler 1" hello.out
```

To run a GDB command after running `hello.out`, use the `-A` option:

```
sh4xrun -t stmc:mb411:st40 -A "stmcprofiler 0" hello.out
```

In both the previous examples, `stmcprofiler` is a user-supplied GDB command that configures the ST Micro Connect profiler.

To run `hello.out` on a silicon target using a GDB configuration script instead of an ST TargetPack, enter the following command:

```
sh4xrun -c mb411bypass -t stmc -e hello.out
```

To run `hello.out` on the ST40 simulator, enter the following command:

```
sh4xrun -c mb411sim -e hello.out
```

To run `hello.out` using a command script, enter the following command:

```
sh4xrun -i run.cmd
```

Where the contents of the script file, `run.cmd`, could be:

```
file hello.out
mb411bypass stmc
load
c
```

To run `hello.out` with target program arguments, enter the following command:

```
sh4xrun -c mb411bypass -t stmc -e hello.out -a arg1 arg2 arg3 arg4
```

5 Using STWorkbench

This chapter describes how to use the STWorkbench Integrated Development Environment (IDE) for the ST40 Micro Toolset. STWorkbench is available on all supported host platforms.

The STWorkbench is delivered with CDT (C/C++ Development Tooling) included. CDT provides a fully functional C and C++ IDE for the STWorkbench platform and enables the user to develop, execute and debug applications interactively.

The STWorkbench is built on the Eclipse IDE. The Eclipse development environment and related information can be found at the Eclipse website http://www.eclipse.org. Information on CDT can be found at http://www.eclipse.org/cdt.

Note: *STWorkbench is a separate release to the ST40 Micro Toolset. Contact your STMicroelectronics FAE or ST support center for more information.*

5.1 Getting started with STWorkbench

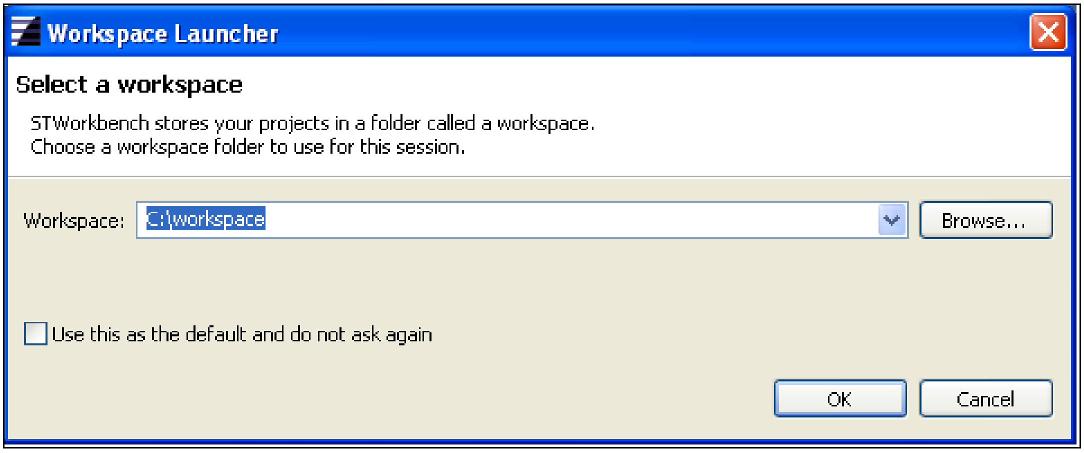
Under Linux, start STWorkbench from the shell by entering `stworkbench`.

Under Windows, start STWorkbench by selecting the appropriate option from the Start menu: **Programs > STM Tools > STWorkbench R*n.n.n* > STWorkbench**, where *n.n.n* is the STWorkbench version number.

Note: *The precise menu options displayed are dependant upon the version of STWorkbench in use.*

When STWorkbench is launched, the **Workspace Launcher** dialog is displayed (see *Figure 2*). Use this dialog to enter or select the location of the workspace. The workspace is the directory where the project data, files and directories are stored.

Figure 2. Workspace Launcher



Workspace Launcher

Select a workspace

STWorkbench stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: C:\workspace

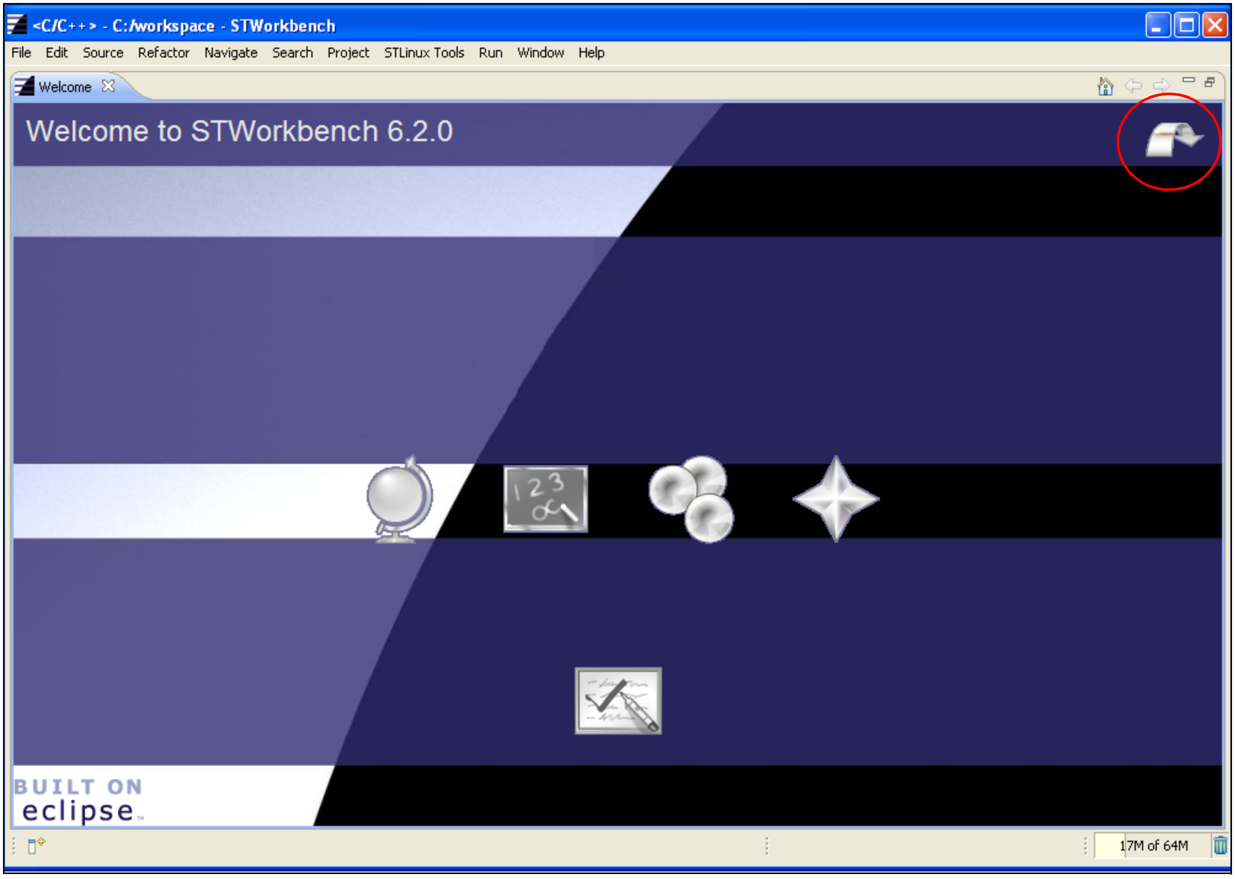
If the workspace directory does not already exist, STWorkbench creates it.

Note: 1 *Do not use spaces in the workspace path and name as it may cause problems with the tools.*

2 *The workspace can be changed at any time by selecting **Switch Workspace** from the **File** menu.*

When STWorkbench is launched for the first time, the **C/C++ Projects** perspective is displayed, with only the **Welcome to STWorkbench** view visible. See *Figure 3*.

Figure 3. Welcome view


<C/C++> - C:/workspace - STWorkbench
File Edit Source Refactor Navigate Search Project STLinux Tools Run Window Help
Welcome
Welcome to STWorkbench 6.2.0
1 2 3
BUILT ON
eclipse
17M of 64M

The icons on this screen allow access to documentation about STWorkbench, tutorials and a registration wizard. If you are a first-time user, take some time to explore the documentation to learn more about STWorkbench.

Note: If you have not done so already, STMicroelectronics recommend that you register your installation of STWorkbench. This will add your name to a mailing list that provides help and advice on using STWorkbench. You will be able to unsubscribe from the mailing list at any time.

Proceed from the **Welcome** view to the **Workbench** by clicking on the curved arrow icon in the top right corner of the Welcome screen, circled in red in *Figure 3*. You can return to the **Welcome** view at any time by selecting **Help > Welcome**.

A **Workbench** provides one or more perspectives. A perspective contains editors and views, such as the **Navigator**. Multiple **Workbenches** can be opened simultaneously.

5.1.1 The STWorkbench workbench

Before using STWorkbench, it is important to become familiar with the various elements of the workbench. A workbench consists of:

- perspectives
- views
- editors

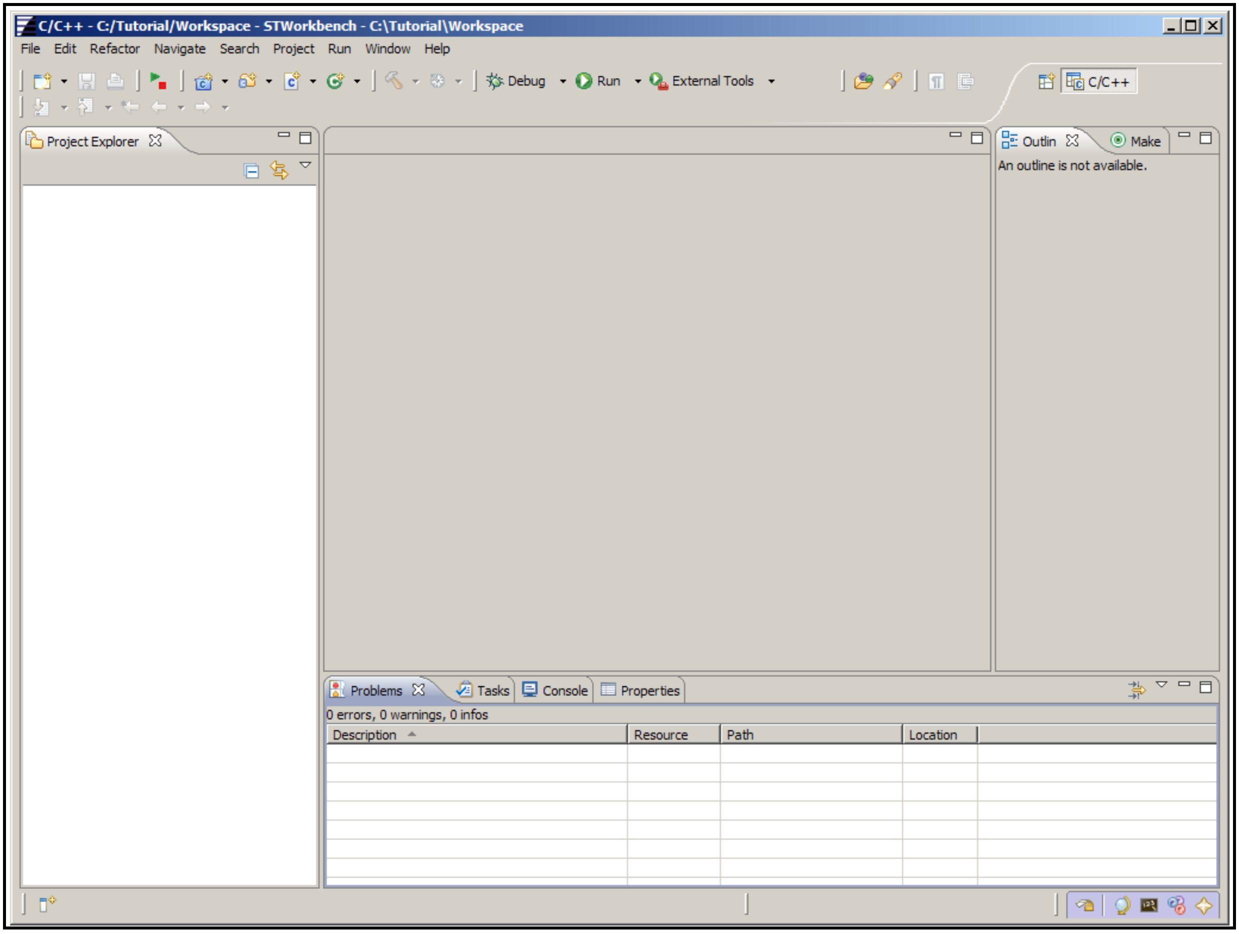
A perspective is a predefined group of views and editors in the **Workbench**. A perspective is designed to include all the views necessary for carrying out a specific task. For example, the **C/C++** perspective contains views required for C/C++ development (including the **C/C++ Projects** view and the **Outline** view) and the **Debug** perspective contains views required when debugging (including the **Debug**, **Variables** and **Breakpoints** views). One or more perspectives can exist in a single workbench. Each perspective contains one or more views and editors. Each perspective may have a different set of views but all perspectives share the same set of editors.

A view is a window within the workbench. It is typically used to navigate through a hierarchy of information (such as the resources in the workbench), open an editor, or display properties for the active editor. Modifications made in a view are saved immediately.

Several views in the Debug perspective can be duplicated to show multiple views of the same type of information, with each locked into different contexts in the Debug view. For more information, launch the STWorkbench help system with **Help > Help Contents** and then see **STWorkbench Help > Pin and Clone**.

The title bar of the **Workbench** indicates which perspective and workspace is active. In *Figure 4*, the **C/C++ Projects** perspective is in use, and the workspace is located at **C:\Tutorial\Workspace**.

Figure 4. C/C++ perspective


C/C++ - C:/Tutorial/Workspace - STWorkbench - C:\Tutorial\Workspace
File Edit Refactor Navigate Search Project Run Window Help
Debug
Run
External Tools
C/C++
Project Explorer
Outlin
Make
Problems
Tasks
Console
Properties

An outline is not available.

0 errors, 0 warnings, 0 infos

Description	Resource	Path	Location

An editor is a visual component within the workbench. It is typically used to edit or browse a resource. Multiple instances of an editor may exist within a workbench window.

Depending on the type of file being edited, the appropriate editor appears in the editor area. For example, if a `.txt` file is being edited, a text editor appears. The name of the file appears in the editor tab. If an asterisk (`*`) appears on the left of the tab, it shows the editor has unsaved changes. If you try to close the editor or exit the workbench without saving, a prompt to save the editor's changes appears.

When an editor is active, the workbench menu bar and toolbar contain operations applicable to the editor. When a view becomes active, the editor operations are disabled. However, certain operations may be appropriate in a view and remain enabled.

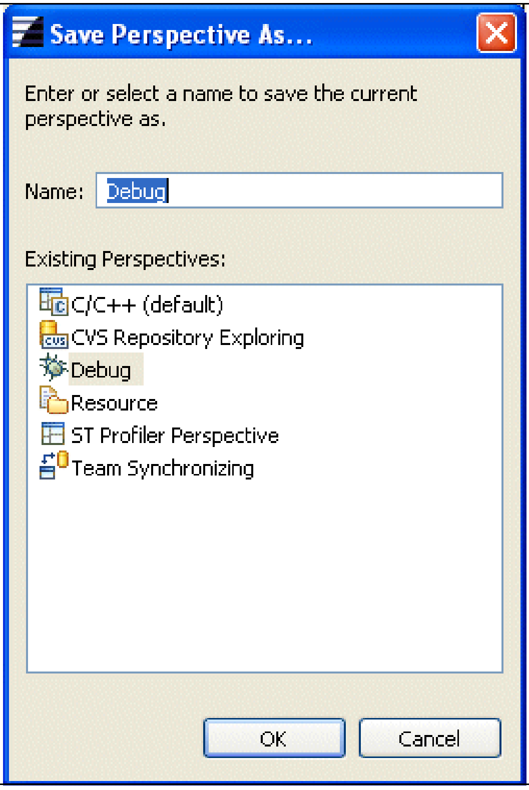
The editors can be stacked in the editor area. Click the tab for a particular editor to use it. Editors can also be tiled side-by-side in the editor area so their content can be viewed simultaneously.

Changing a perspective’s views

The views that make up a perspective can be changed. For example, to add the **Disassembly** view to the **Debug** perspective, complete the following steps.

1. If necessary, change to the **Debug** perspective by selecting **Window > Open Perspective > Debug** or **Window > Open Perspective > Other... > Debug**.
2. Select **Window > Show View > Disassembly** to display the **Disassembly** view as part of this perspective.
3. Select **Window > Save Perspective As...**. The **Save Perspective As...** dialog is displayed.

Figure 5. Save Perspective As... dialog



Save Perspective As...

Enter or select a name to save the current perspective as.

Name: Debug

Existing Perspectives:

- C/C++ (default)
- CVS Repository Exploring
- Debug
- Resource
- ST Profiler Perspective
- Team Synchronizing

4. Select **Debug** in the **Existing Perspectives** list and click on **OK**.

   You are prompted:

   A perspective with the name ‘Debug’ already exists. Do you want to overwrite?
5. Click on **Yes**, to save the **Debug** perspective with the currently open views.

5.2 STWorkbench tutorials and reference pages

The on-line help provides a number of tutorials to guide the user through the steps to build, run and debug an ST40 application. The tutorials are accessed through the STWorkbench help system by selecting **Help > Help Contents > STWorkbench for OS21 and STLinux**.

On completion of each of the building and importing tutorials, you will have built an ST40 application ready to run or debug.

Introduction > Getting Started

This set of tutorials provides instructions on how to build, debug and run a simple OS21 “Hello World” C application. The debugging tutorial covers common debugging operations such as modifying breakpoints, examining variables, call stacks and tasks.

Introduction > STWorkbench concepts

The purpose of this tutorial is to familiarize new users with the various basic elements of the workbench.

Building > Building a C/C++ application with an automatically generated makefile

This tutorial describes how to use STWorkbench’s Executable Project (formerly Managed Build System) to automatically create the makefile for a simple OS21 application.

Building > Building a C/C++ application with a makefile

This tutorial describes how to build a simple application using STWorkbench’s Makefile project (formerly Standard Build System) facility. This tutorial requires the user to create a makefile, which STWorkbench uses to build the application.

Building > Building an existing C/C++ application by importing the makefile

This tutorial describes the process of importing existing source code, complete with makefile, into an STWorkbench project. This tutorial requires the user to create a makefile, which STWorkbench uses to build the application.

Editing > Navigation

This page describes the source code index and search facilities of STWorkbench.

Execute from Command Line

STWorkbench supports the ability to launch a debug session directly from the command line. This set of pages describes how to use this facility.

System Monitoring > Branch Trace View

This page describes how to use the **Branch Trace** view to display the branches that the program performed before arriving at the place where the debugger is currently stopped.

System Monitoring > Performance Counters

This page describes how to use the **ST40 Performance Counters** view to examine the data provided by the ST40 Performance Counters facility.

5.3 ST40 System Analysis tutorials and reference pages

There are several tutorials on how to use the ST System Analysis (formerly the Profiling and Coverage) features. The tutorials are accessed through the STWorkbench help system by selecting **Help > Help Contents > STWorkbench for OS21 and STLinux**.

Profiling and Coverage > ST40 Trace, Profile and Coverage

STWorkbench supports the generation and display of profiling, coverage and OS21 Profiler data. These features are described in the following tutorials:

- **STgprof**

  This tutorial describes the **STgprof** profiler and provides a guide on using this tool to determine which parts of a program take most of the execution time.
- **STgcov**

  This tutorial describes the **STgcov** profiler and provides a guide on using this tool to identify the parts of a program that have never been exercised.
- **OS21 Profiler**

  This tutorial describes the operation of the **OS21 Profiler** profiler to analyze the performance characteristics of an OS21 application. The OS21 Profiler is described in *Chapter 13: Dynamic OS21 profiling on page 214*.

ST40 Interactive Analysis

Interactive support is available for OS21 System Activity, STMC sampling and OS21 Profiler. These features are described in the following help pages.

- **System Trace Viewers > OS21 Activity**

  This tutorial describes the **OS21 Activity** analyzer and provides a guide for using this tool to analyze and monitor the life-cycles of interrupts and tasks in an OS21 application that have been captured using **OS21Trace**. OS21 Trace is described in *Chapter 12: OS21 Trace on page 163*.
- **Profiling and Coverage > STMC sample profiler**

  These pages describe the **STMC sample profiler**, a facility that uses the ST Micro Connect to obtain profiling data on the application. The generated result is similar to **STgprof**. The STMC sample profiler is described in *Appendix D: Profiler plugin on page 270*.
- **System Monitoring > STMC sample history**

  These pages describe the **STMC sample history**, which uses the same approach as the **STMC sample profiler** but provides a sequential view of the application’s activities over a period of time.

6 Using Insight

Insight is a Graphical User Interface for GDB available on all supported host platforms. It enables the user to execute and debug applications interactively. The command line interface for GDB is described in *Section 4.2: The GNU debugger on page 54*.

Insight can display several windows containing source and assembly level code together with a range of system information. In addition, Insight has a **Console Window** for entering GDB commands on the command line.

Insight has the following features.

- Many parts of the window have a context sensitive menu, which is displayed by clicking the right-hand mouse button.
- A tooltip is displayed when the mouse pointer hovers over a button.
- When Insight launches, it restores the configuration and open windows from the state saved in the user’s home directory (specified by the `HOME` environment variable) in a file named `.gdbtkinit` on Linux, or `gdbtk.ini` on Windows. This state is saved whenever the Insight GUI is closed.

6.1 Launching Insight

To launch the Insight GUI, issue either of the following commands:

```
sh4gdb -w
sh4insight
```

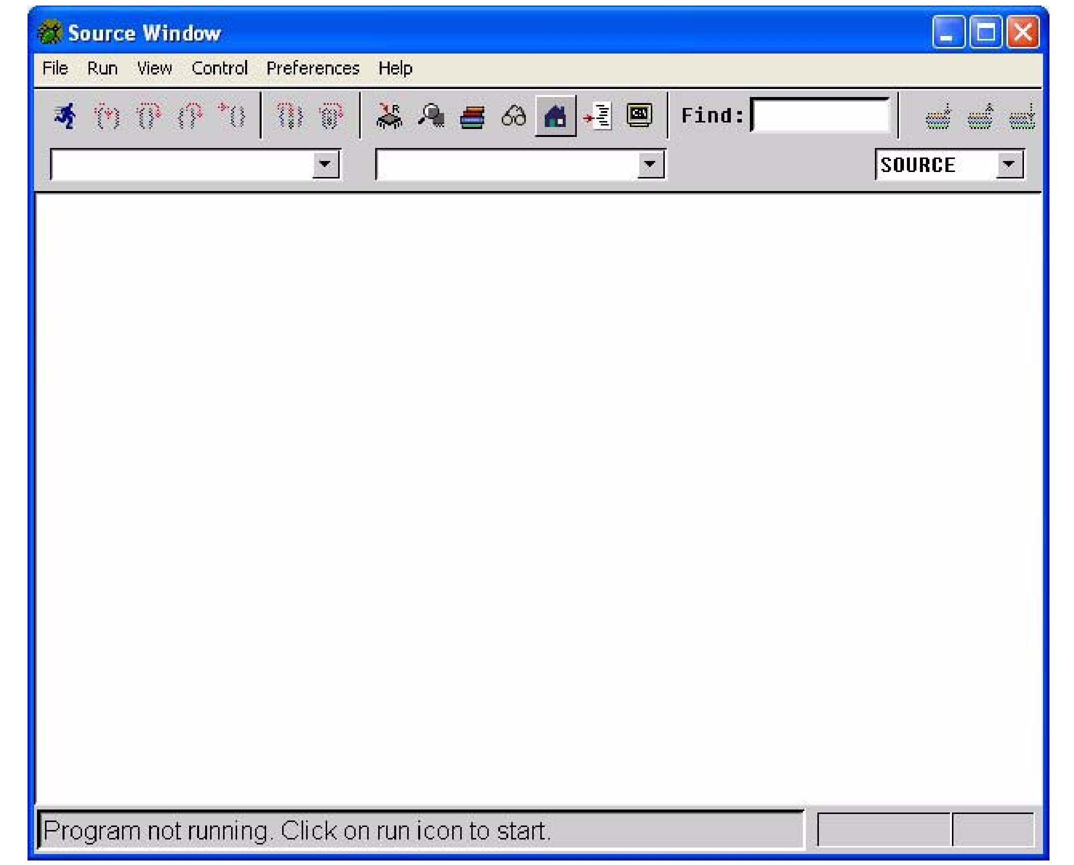
Under Windows, Insight can also be launched by clicking on the **Start** button and selecting **Programs** > **STM Tools > ST40 Micro Toolset R*n.n.n*** > **Insight**, where *n.n.n* is the ST40 Micro Toolset version number.

When Insight is launched for the first time, the **Source Window** is displayed. This window is described in *Section 6.2*.

6.2 Using the Source Window

The **Source Window** is the main window that is displayed when Insight is launched (see *Figure 6*). The menus on the menu bar are described in *Section 6.2.1* and the toolbar buttons are described in *Section 6.2.2*.

Figure 6. Source Window



Program not running. Click on run icon to start.

6.2.1 Source Window menus

File menu

Open... Load a program executable.

Close Close the program executable.

Source... Select a GDB command script to source.

Page Setup... Display a dialog to select the paper size, the paper source and page orientation (landscape or portrait).

Print Source... Display a dialog to select the printer, what to print and the number of copies to be printed.

Target Settings... Display the **Target Selection** window to select and configure the target.

Exit	Close the Insight GUI.

Run menu

Connect to target	Connect to the selected target. If no target is selected, the **Target Selection** window is displayed so that the target can be set up as required.
Download	Download the program to the target.
Run	Download and execute the program. If no target is selected, the **Target Selection** window is displayed so that the target can be set up as required.
Disconnect	Close the connection to the target.

View menu

Stack	Display the **Stack** window.
Registers	Display the **Registers** window.
Memory	Display the **Memory** window.
Watch Expressions	Display the **Watch Expressions** window.
Local Variables	Display the **Local Variables** window.
Breakpoints	Display the **Breakpoints** window.
Console	Display the **Console Window**.
Function Browser	Display the **Function Browser** window.
Thread List	Display the **Processes** window.

Control menu

Step	Step into the next statement.
Next	Step over the next statement.
Finish	Step out of the current function.
Continue	Continue the program after a breakpoint.
Step Asm Inst	Step one instruction.
Next Asm Inst	Step one instruction and proceed through subroutine calls.

Preferences menu

Global...	Display the **Global Preferences** window.
Source...	Display the **Source Preferences** window.

Advanced Displays the Advanced submenu. This menu has two options: **Edit Color Schemes** defines colors that can be used as text backgrounds, and **IPC Support** enables the IPC feature that can be used to control multiple instances of Insight.

Use Color Scheme Selects the color scheme to use in Insight. 16 color schemes can be defined. They are modified by selecting **Edit Color Schemes** from the **Advanced** menu.

Help menu

Help Topics Display the online help window.

About GDB... Display version and copyright information for the Insight GUI.

6.2.2 Source Window toolbar

Table 20 provides a brief explanation of each of the buttons available on the **Source Window** toolbar.

Table 20. The Source Window buttons

Button	Name	Description
🏃	Run (R)	Start the program executing.
{↓}	Step (S)	Step into the next statement.
{}↓	Next (N)	Step over the next statement.
{}↓	Finish (F)	Step out of the current function.
→{}	Continue (C)	Continue the program after a breakpoint.
{i}	Step Asm Inst (S)	Step one instruction.
{i}↓	Next Asm Inst (N)	Step over the next instruction.
R	Registers (Ctrl+R)	Display the **Registers** window.
🔍	Memory (Ctrl+M)	Display the **Memory** window.
☰	Stack (Ctrl+S)	Display the **Stack** window.
👓	Watch Expressions (Ctrl+W)	Display the **Watch Expressions** window.

Table 20. The Source Window buttons (continued)

Button	Name	Description
	Local Variables (Ctrl+L)	Display the **Local Variables** window.
	Breakpoints (Ctrl+B)	Display the **Breakpoints** window.
	Console (Ctrl+N)	Display the **Console Window**.
	Down Stack Frame	Move to the stack frame called by the current frame.
	Up Stack Frame	Move to the stack frame that called the current frame.
	Go To Bottom of Stack	Move to the bottom most stack frame.

6.2.3 Context sensitive menus

Many parts of a window have context sensitive menus. To open a context sensitive menu, click the right-hand mouse button.

For example, right-click in the left margin of any line in the source code where a breakpoint can be set (indicated by a hyphen) to display a context sensitive menu containing the options listed below.

Continue to Here Continue the application and stop at the selected line.

Jump to Here Jump directly to the specified line[1]. This does not operate in the same way as the Continue option since only the Program Counter is modified. This option is advantageous for running a given code sequence a second time after the contents of a variable has been manually modified, or for skipping over defective code.

Set Breakpoint Set a breakpoint on the selected line. The breakpoint is displayed as a red square.

Set Temporary Breakpoint Set a temporary (one time only) breakpoint on the selected line. The breakpoint is displayed as an orange square.

Set Breakpoint on Thread(s)...

Set a breakpoint on the thread. If more than one thread is available the **Thread Selection** window is displayed to select the required threads. The breakpoint is displayed as a pink square.

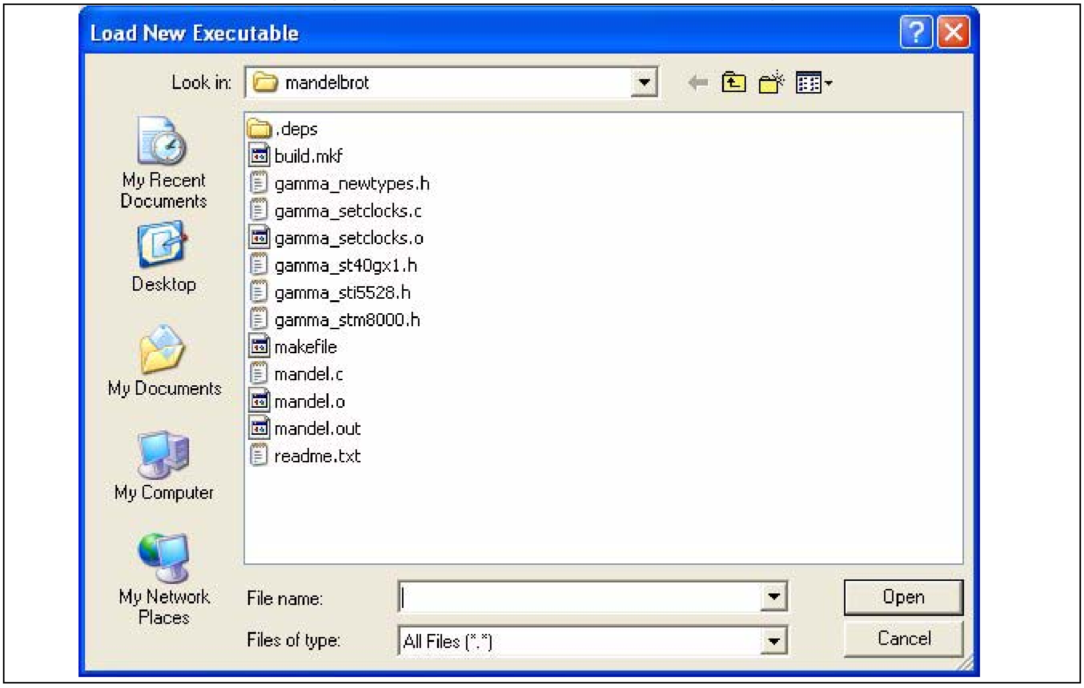
[1] In optimized code, this may not work as expected due to the compiler reordering code.

6.3 Debugging a program

The following procedure demonstrates debugging a program using the **mandelbrot** example, see *Section 1.5.3: The examples directory on page 28*.

1. Launch Insight as described in *Section 6.1 on page 75*.
2. Click on or select **Run** from the **Run** menu. The **Load New Executable** dialog opens. See *Figure 7*.

Figure 7. Load New Executable dialog



Load New Executable

Look in: mandelbrot

.deps
build.mkf
gamma_newtypes.h
gamma_setclocks.c
gamma_setclocks.o
gamma_st40gx1.h
gamma_sti5528.h
gamma_stm8000.h
makefile
mandel.c
mandel.o
mandel.out
readme.txt

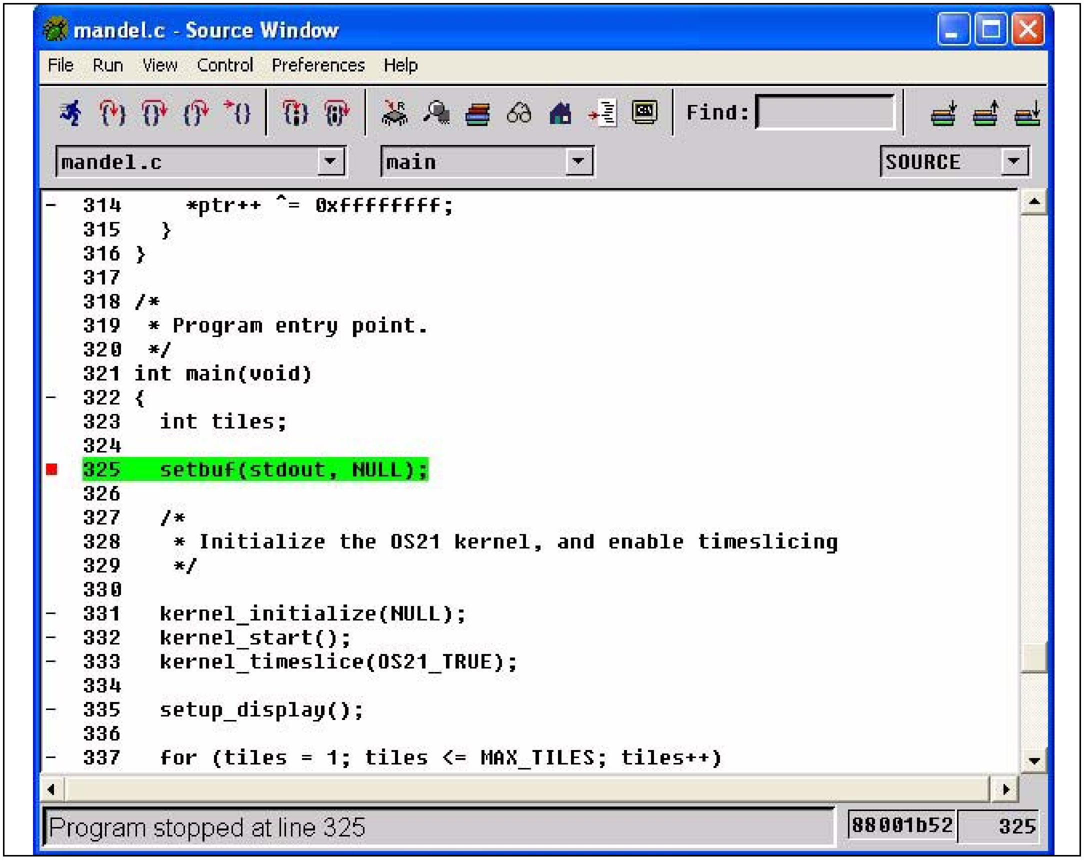
File name:

Files of type: All Files (*.*)

3. Select the executable file and click on **Open**. The **Target Selection** window opens.

4. Complete this window as described in *Section 6.4: Changing the target*. The program is launched and stopped at the breakpoint set at the `main()` function. See *Figure 8*.

Figure 8. mandel.c stopped at breakpoint



```
    *ptr++ ^= 0xffffffff;
  }
}

/*
 * Program entry point.
 */
int main(void)
{
  int tiles;

  setbuf(stdout, NULL);

  /*
   * Initialize the OS21 kernel, and enable timeslicing
   */

  kernel_initialize(NULL);
  kernel_start();
  kernel_timeslice(OS21_TRUE);

  setup_display();

  for (tiles = 1; tiles <= MAX_TILES; tiles++)
```

```
Program stopped at line 325    88001b52    325
```

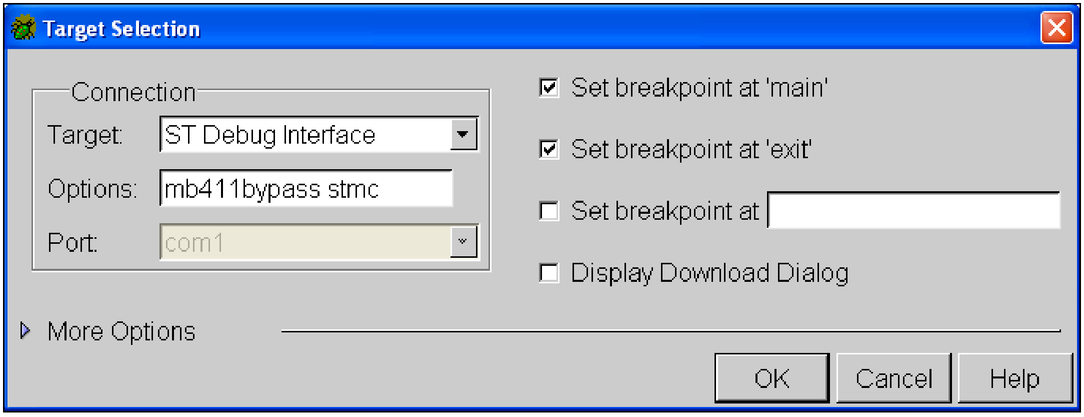
5. Debug the program using the menu and toolbar options as described in *Section 6.2.2 on page 78* and *Section 6.2.2 on page 78*.

   To toggle breakpoints on and off, click on the hyphen symbols to the left of the code. Breakpoints are shown as red squares.

6.4 Changing the target

1. Select **Target Settings...** from the **File** menu. The **Target Selection** window opens, see *Figure 9*.

Figure 9. Target Selection window



Target Selection

Connection

Target: ST Debug Interface

Options: mb411bypass stmc

Port: com1

☑ Set breakpoint at 'main'

☑ Set breakpoint at 'exit'

□ Set breakpoint at

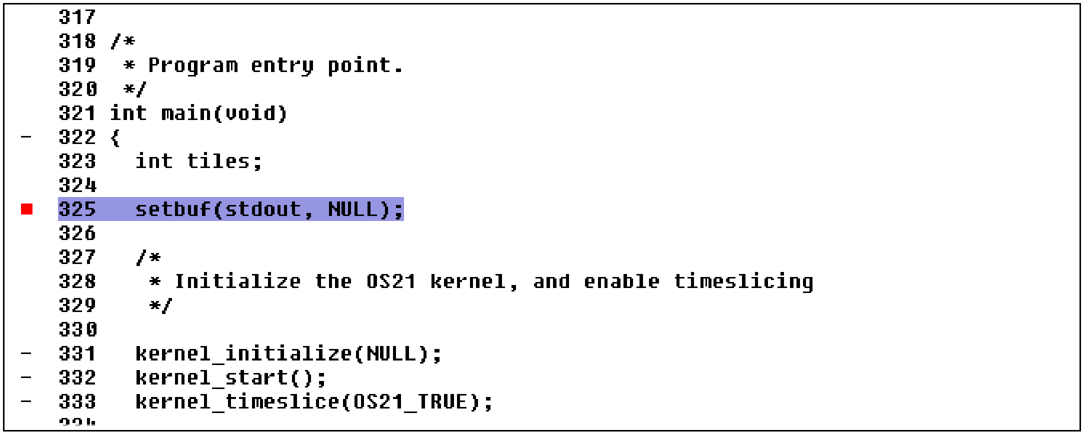
□ Display Download Dialog

2. From the **Target** drop-down list, select **ST Debug Interface**.
3. Specify any **Options** required, for example, enter `mb411bypass stmc` to run the example on an STb7100-MBoard connected to an ST Micro Connect with a name of `stmc`, or enter `mb411sim` to run the example on the ST40 simulator configured as an STb7100-MBoard.
4. Click on **OK**.

6.5 Configuring breakpoints

When a program runs, it continues as far as the first breakpoint. If **Set breakpoint at 'main'** in the **Target Selection** window is selected, this is the first real line of the program. See *Figure 10*.

Figure 10. Breakpoint examples



```
  
  /*
   * Program entry point.
   */
  int main(void)
- {
    int tiles;
  
■   setbuf(stdout, NULL);
  
    /*
     * Initialize the OS21 kernel, and enable timeslicing
     */
  
-   kernel_initialize(NULL);
-   kernel_start();
-   kernel_timeslice(OS21_TRUE);
```

The red square in the left-hand margin indicates where a breakpoint has been set. The hyphens indicate valid positions for potential breakpoints.

The colored background to line 325 indicates the position of the current Program Counter (PC). Orange highlighting indicates the current position in that stack frame (the real position is at the top of the stack).

When the mouse pointer hovers over a variable or function name, a tooltip shows the current value of that variable. Variables and types have a context sensitive menu (available by right-clicking on the item) that has various actions, for example, setting watchpoints and dumping memory.

To set a breakpoint, click on the hyphen next to the line of code. The breakpoint is displayed as a red square.

Right click on a breakpoint position (shown as a hyphen) to display the context sensitive menu containing the options listed below to configure breakpoints.

Set Breakpoint Set a breakpoint on the selected line. The breakpoint is shown as a red square.

Set Temporary Breakpoint Set a temporary (one time only) breakpoint on the selected line. The breakpoint is shown as an orange square.

Set Breakpoint on Thread(s)... Set a breakpoint on the thread. If more than one thread is available the **Thread Selection** window is displayed to select the required threads. The breakpoint is displayed as a pink square.

To replace the three **Set Breakpoint** options with **Disable Breakpoint** and **Delete Breakpoint** options, right-click on an existing breakpoint. Disabled breakpoints are displayed as black squares.

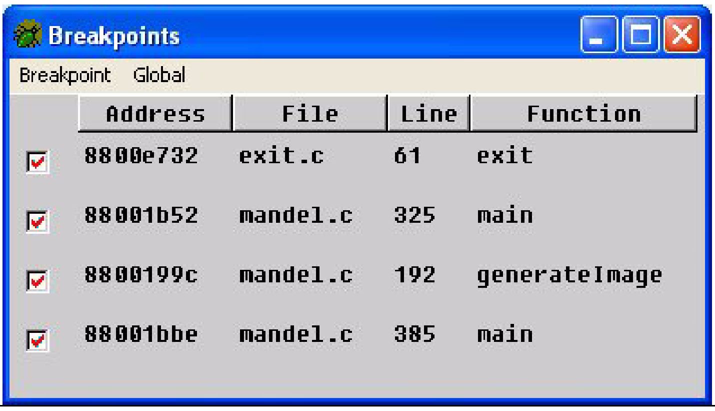
6.5.1 The Breakpoints window

Breakpoints can also be controlled using the **Breakpoints** window (*Figure 11*). To open the **Breakpoints** window, either:

- click on , or
- select **Breakpoints** from the **View** menu in the **Source Window**

Note: *The **Breakpoints** window does not allow the creation of new breakpoints, but does permit existing ones to be viewed and edited.*

Figure 11. Breakpoints window



	Address	File	Line	Function
☑	8800e732	exit.c	61	exit
☑	88001b52	mandel.c	325	main
☑	8800199c	mandel.c	192	generateImage
☑	88001bbe	mandel.c	385	main

Click on a breakpoint to select it. To change the breakpoint, use the check boxes and the **Breakpoint** and **Global** menus.

6.6 Using the help

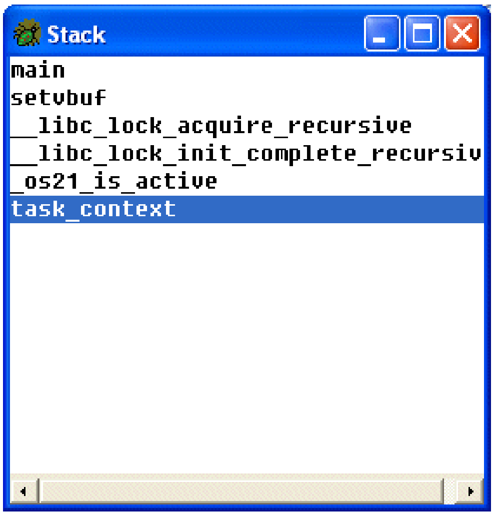
Additional information is available in the help files supplied with the Insight GUI. To access the help files, select **Help Topics** from the **Help** menu.

6.7 Using the Stack window

The **Stack** window (*Figure 12*) contains a list of all the frames currently on the stack. To open the **Stack** window, either:

- click on ☰, or
- select **Stack** from the **View** menu in the **Source Window**.

Figure 12. Stack window



```
main
setvbuf
__libc_lock_acquire_recursive
__libc_lock_init_complete_recursiv
_os21_is_active
task_context
```

To select a frame, click on the appropriate frame line. The line is highlighted in yellow and the **Registers** and **Local Variables** windows show the associated data. The **Source Window** shows the associated source line. See *Figure 13 on page 85*, *Figure 18 on page 89* and *Figure 6 on page 76* respectively.

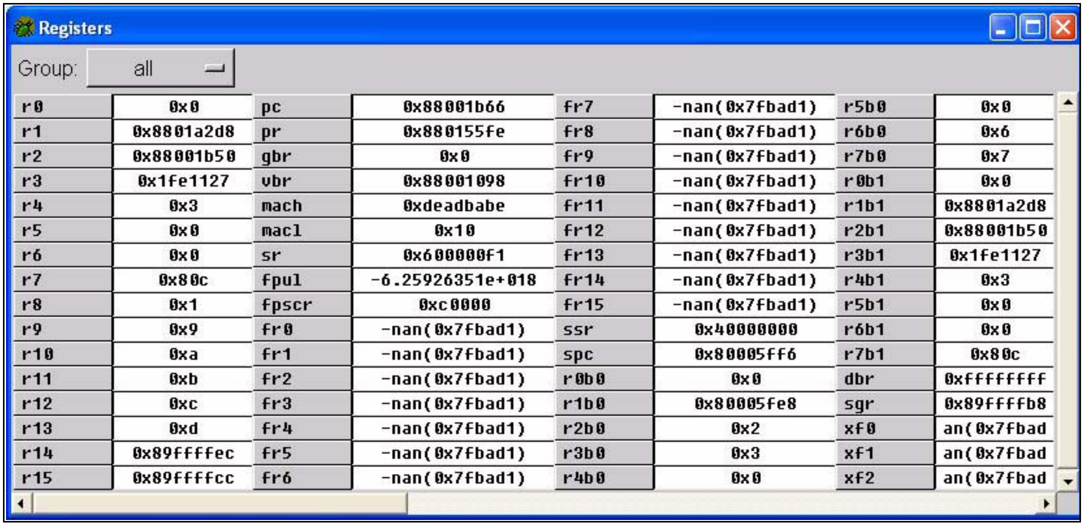
6.8 Using the Registers window

The **Registers** window (*Figure 13*) displays the contents of all the registers.

To open the **Registers** window, either:

- click on , or
- select **Registers** from the **View** menu in the **Source Window**.

Figure 13. Registers window



Registers

Group: all

r0	0x0	pc	0x88001b66	fr7	-nan(0x7fbad1)	r5b0	0x0
r1	0x8801a2d8	pr	0x880155fe	fr8	-nan(0x7fbad1)	r6b0	0x6
r2	0x88001b50	gbr	0x0	fr9	-nan(0x7fbad1)	r7b0	0x7
r3	0x1fe1127	vbr	0x88001098	fr10	-nan(0x7fbad1)	r0b1	0x0
r4	0x3	mach	0xdeadbabe	fr11	-nan(0x7fbad1)	r1b1	0x8801a2d8
r5	0x0	macl	0x10	fr12	-nan(0x7fbad1)	r2b1	0x88001b50
r6	0x0	sr	0x600000f1	fr13	-nan(0x7fbad1)	r3b1	0x1fe1127
r7	0x80c	fpul	-6.25926351e+018	fr14	-nan(0x7fbad1)	r4b1	0x3
r8	0x1	fpscr	0xc0000	fr15	-nan(0x7fbad1)	r5b1	0x0
r9	0x9	fr0	-nan(0x7fbad1)	ssr	0x40000000	r6b1	0x0
r10	0xa	fr1	-nan(0x7fbad1)	spc	0x80005ff6	r7b1	0x80c
r11	0xb	fr2	-nan(0x7fbad1)	r0b0	0x0	dbr	0xffffffff
r12	0xc	fr3	-nan(0x7fbad1)	r1b0	0x80005fe8	sgr	0x89ffffb8
r13	0xd	fr4	-nan(0x7fbad1)	r2b0	0x2	xf0	an(0x7fbad
r14	0x89ffffec	fr5	-nan(0x7fbad1)	r3b0	0x3	xf1	an(0x7fbad
r15	0x89ffffcc	fr6	-nan(0x7fbad1)	r4b0	0x0	xf2	an(0x7fbad

Click on a register to select it. A register value can be modified by editing its value in the **Registers** window. A register can be operated upon by right clicking on it to display the context sensitive menu containing the options.

Hex, Decimal, Unsigned Change the format in which the information is displayed.

Open Memory Window Open a **Memory** window at the location specified by the currently selected register. See *Section 6.9 on page 86*.

Add to Watch Add the selected register to the **Watch** window. See *Section 6.10 on page 88*.

Remove from Display Delete the selected register from the window.

Display all Registers Restore all registers that have been removed from the display.

Help Displays the online help window.

Close Close the **Registers** window.

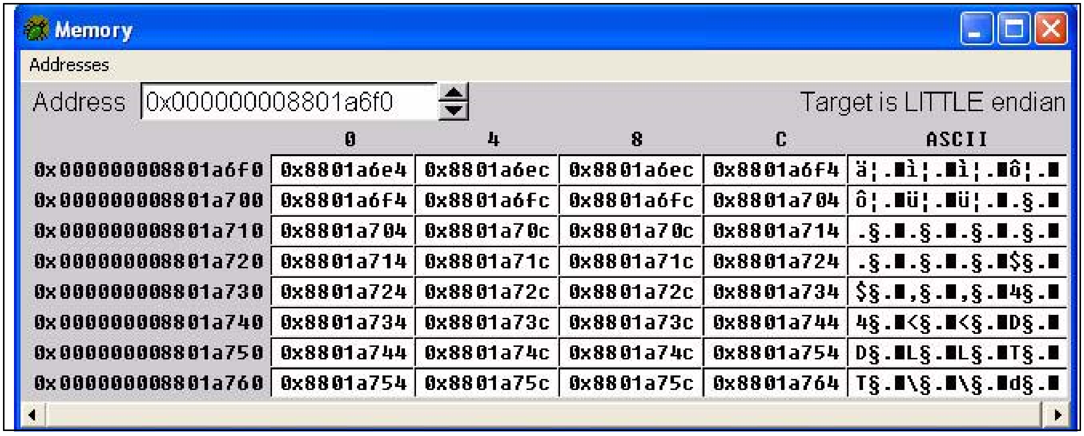
*Note: To view only the registers belonging to a specific group (**general**, **float**, **system**, **vector**, **all**), use the **Group** selection box.*

6.9 Using the Memory window

The **Memory** window (*Figure 14*) allows the current state of memory on the target to be viewed and modified. The window can be resized to view more memory information. To open the **Memory** window, either:

- click on , or
- select **Memory** from the **View** menu in the **Source Window**.

Figure 14. Memory window

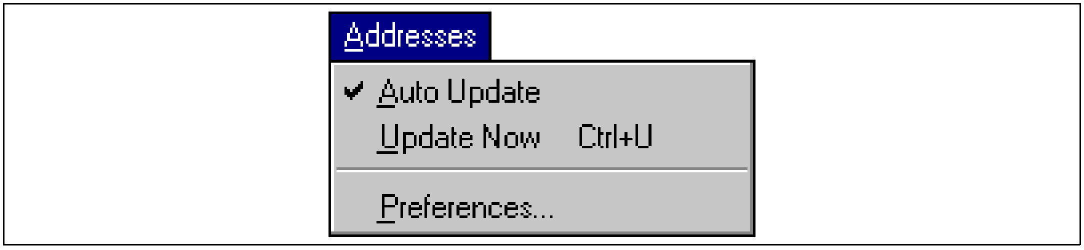


Address 0x000000008801a6f0 Target is LITTLE endian

	0	4	8	C	ASCII
0x000000008801a6f0	0x8801a6e4	0x8801a6ec	0x8801a6ec	0x8801a6f4	ä¦.■ì¦.■ì¦.■ô¦.■
0x000000008801a700	0x8801a6f4	0x8801a6fc	0x8801a6fc	0x8801a704	ô¦.■ü¦.■ü¦.■.§.■
0x000000008801a710	0x8801a704	0x8801a70c	0x8801a70c	0x8801a714	.§.■.§.■.§.■.§.■
0x000000008801a720	0x8801a714	0x8801a71c	0x8801a71c	0x8801a724	.§.■.§.■.§.■$§.■
0x000000008801a730	0x8801a724	0x8801a72c	0x8801a72c	0x8801a734	$§.■,§.■,§.■4§.■
0x000000008801a740	0x8801a734	0x8801a73c	0x8801a73c	0x8801a744	4§.■<§.■<§.■D§.■
0x000000008801a750	0x8801a744	0x8801a74c	0x8801a74c	0x8801a754	D§.■L§.■L§.■T§.■
0x000000008801a760	0x8801a754	0x8801a75c	0x8801a75c	0x8801a764	T§.■\§.■\§.■d§.■

Click on a memory location to amend the contents. To customize the display, use the **Addresses** menu (*Figure 15*).

Figure 15. Addresses menu



Auto Update If the state of the target changes, the memory information is updated automatically. (Default.)

Update Now Manually override the auto-update to display the memory state at that instant.

Preferences... Display the Memory Preferences window (see *Figure 16*).

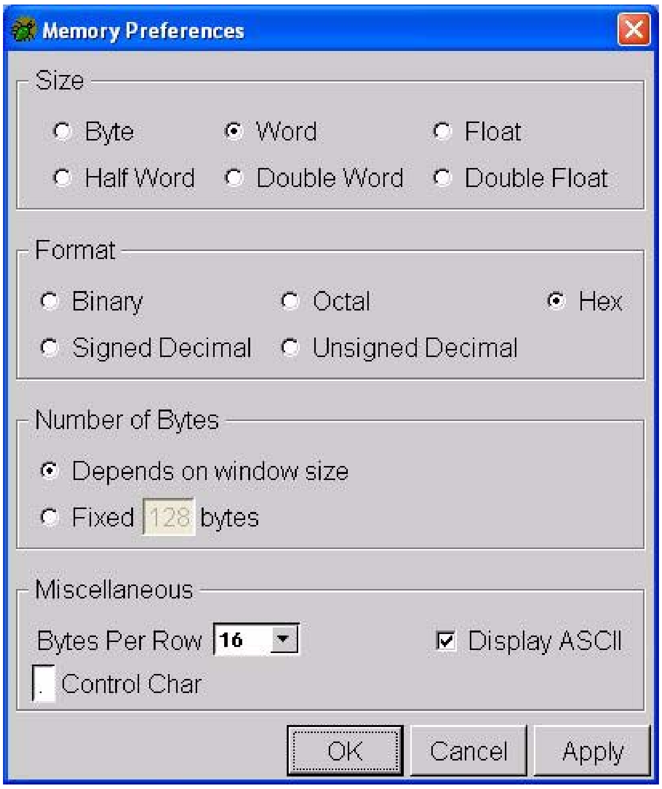
This window can be used to select the size of the cells, format the memory display, select the number of bytes to be displayed, select or enter (and then press **Return**) the number of bytes per row, select whether to display the memory as ASCII text, and select the character to use for non-ASCII characters (normally ‘ . ’).

Right click on a memory location to open the following context-sensitive menu options:

Go To ... Display the selected memory location.

Open New Window at ... Open an additional **Memory** window displaying the selected memory location.

Figure 16. Memory Preferences window



Memory Preferences

Size

○ Byte ● Word ○ Float

○ Half Word ○ Double Word ○ Double Float

Format

○ Binary ○ Octal ● Hex

○ Signed Decimal ○ Unsigned Decimal

Number of Bytes

● Depends on window size

○ Fixed 128 bytes

Miscellaneous

Bytes Per Row 16 ☑ Display ASCII

. Control Char

6.10 Using the Watch window

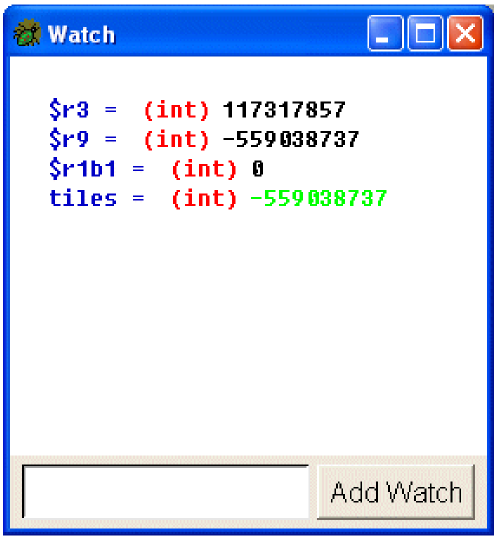
Use the **Watch** window (*Figure 17*) to set and edit user-specified expressions. Each time the program halts, the expressions are reevaluated and the result displayed. This allows the user to see, at a glance, all the program state of interest.

Note: Watch expressions are not the same as watchpoints. Watchpoints must be set through the console window. See Section B.2.4: Silicon specific commands on page 263.

To open the **Watch** window, either:

- click on 👓, or
- select **Watch Expressions** from the **View** menu in the **Source Window**.

Figure 17. Watch window



```
$r3 =  (int) 117317857
$r9 =  (int) -559038737
$r1b1 =  (int) 0
tiles =  (int) -559038737
```

There are two ways to add expressions to the **Watch** window.

- Type an expression into the field at the bottom of the window and click on **Add Watch** (or press **Return**).
- Select the expression In the **Source Window** or **Registers** window, right-click on the expression to open the context-sensitive menu and select **Add to Watch**.

Note: The expression must use the same syntax as the language being debugged. For example, to watch for `i` *being assigned the value* `42` *when debugging a C application, enter* `i==42`*. Using assignment operators by mistake, for example,* `i=42`*, changes the value of the variable in the program.*

Click on a watch expression to select it. It can then be operated upon by right clicking on it to display the context sensitive menu containing the following options:

Format Change the format to **Hex**, **Decimal**, **Binary**, **Octal** or **Natural** (mantissa and exponent for floating-point values).

Edit Edit the expression value.

Delete Delete the highlighted expression from the list.

Dump Memory at ... Displays the selected watch expression in the **Memory** window.

Help Displays the online help window.

Close Close the **Watch** window.

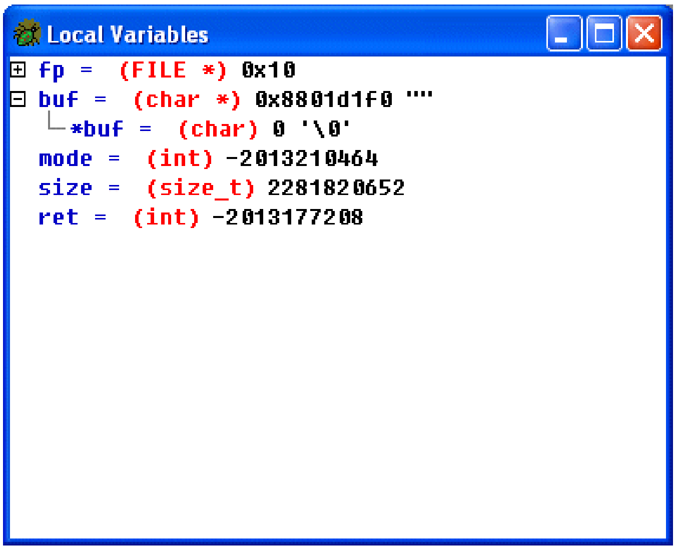
The display of values can also be adjusted by normal C type casting. Structures and classes can be expanded as a tree.

6.11 Using the Local Variables window

The **Local Variables** window (*Figure 18*) shows all the variables in the current stack frame. To open the **Local Variables** window, either:

- click on 🏠, or
- select **Local Variables** from the **View** menu in the **Source Window**.

Figure 18. Local Variables window



```
⊞ fp =  (FILE *) 0x10
⊟ buf =  (char *) 0x8801d1f0 ""
   └*buf =  (char) 0 '\0'
  mode =  (int) -2013210464
  size =  (size_t) 2281820652
  ret =  (int) -2013177208
```

Click on a variable to select it. It can then be amended by right clicking on it to display the context sensitive menu containing the options.

Format Change the format of the variable. It can be **Hex**, **Decimal**, **Octal**, **Binary** or **Natural** (mantissa and exponent for float variables).

Edit Edit the value of the selected variable.

Delete Delete the highlighted expression from the list.

Dump Memory at ... Displays the selected variable in the **Memory** window.

Help Displays the online help window.

Close Close the **Local Variables** window.

To expand the structure of a variable, click on the plus (+) sign. To collapse the structure, click on the minus (-) sign.

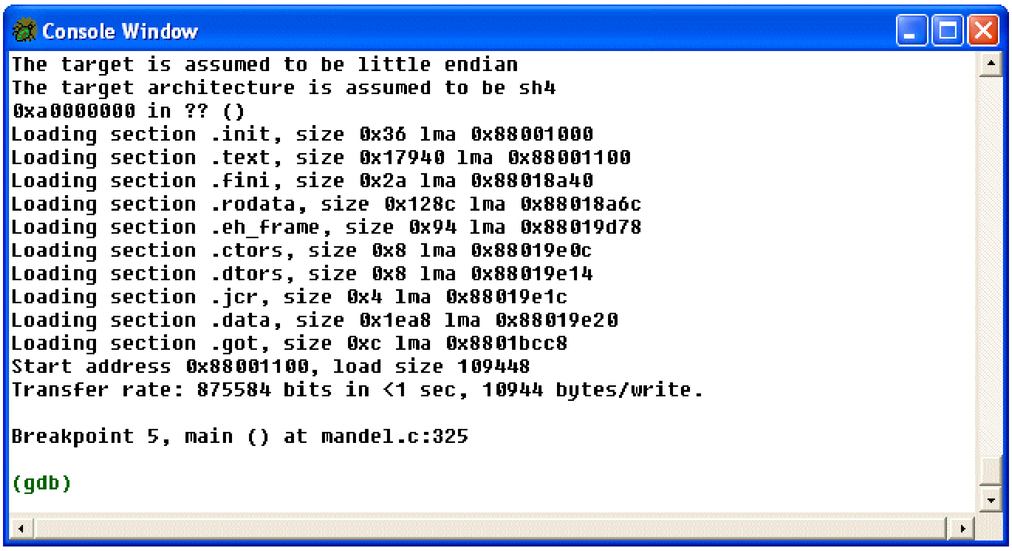
6.12 The Console Window

The **Console Window** (*Figure 19*) is the underlying GDB console and allows commands to be issued directly to GDB.

To open the **Console Window**, either:

- click on , or
- select **Console** from the **View** menu in the **Source Window**.

Figure 19. Console Window



```
The target is assumed to be little endian
The target architecture is assumed to be sh4
0xa0000000 in ?? ()
Loading section .init, size 0x36 lma 0x88001000
Loading section .text, size 0x17940 lma 0x88001100
Loading section .fini, size 0x2a lma 0x88018a40
Loading section .rodata, size 0x128c lma 0x88018a6c
Loading section .eh_frame, size 0x94 lma 0x88019d78
Loading section .ctors, size 0x8 lma 0x88019e0c
Loading section .dtors, size 0x8 lma 0x88019e14
Loading section .jcr, size 0x4 lma 0x88019e1c
Loading section .data, size 0x1ea8 lma 0x88019e20
Loading section .got, size 0xc lma 0x8801bcc8
Start address 0x88001100, load size 109448
Transfer rate: 875584 bits in <1 sec, 10944 bytes/write.

Breakpoint 5, main () at mandel.c:325

(gdb)
```

If the **Console Window** is open when a GDB command is issued, it shows the the output. For example, note the output of the `load` command in *Figure 19*.

*Note: Insight GUI commands such as `continue` or `step` are not visible in the **Console Window** unless they are issued directly at the **Console Window** prompt.*

The display output of the Insight GUI and the GDB console commands are synchronized.

You can issue any GDB command through the **Console Window**.

*Note: 1 The `console off` command directs the program output to the terminal from which Insight was launched and not to the **Console Window**. For this reason, it is better to use `console on` in conjuction with Insight.*

*2 The ST40 simulator instruction trace also appears in the terminal from which Insight was launched and not in the **Console Window**.*

6.13 Function Browser window

To search for functions in the application and show the source code for that function, use the **Function Browser** window (see *Figure 20*). This makes it easy to add breakpoints throughout the code.

To open the **Function Browser** window, from the **View** menu in the **Source Window**, select **Function Browser**. The following fields are available to search for functions.

Function Filter Use this to search with a specified expression.

starts with lists all functions that start with the expression.

contains lists all functions that contain the expression.

ends with lists all functions that end with the expression.

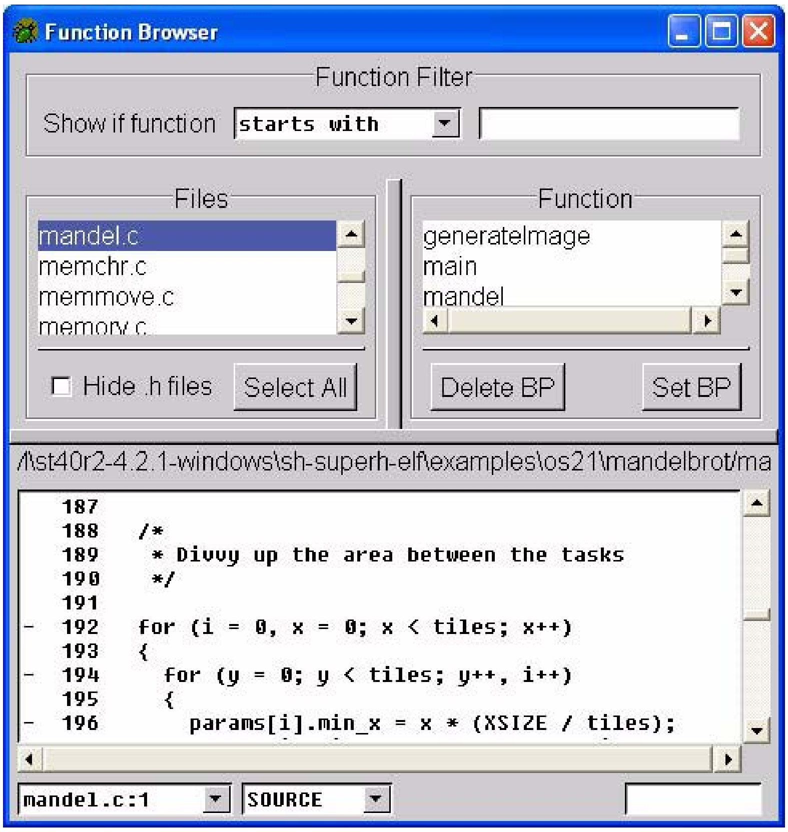
matches regexp lists all functions that match the regular expression.

Files This shows all of the files within the application. Only the selected files are searched for using the expression.

Functions This shows all the functions within the selected files. To delete and set breakpoints at the start of each function, use the **Delete BP** and **Set BP** buttons.

The lower section of the window shows the source code for the selected function. To set breakpoints, use the same method as for the **Source Window**, see *Section 6.5 on page 82*.

Figure 20. Function Browser window



Show if function starts with

Files: mandel.c, memchr.c, memmove.c, memory.c

Function: generateImage, main, mandel

/\st40r2-4.2.1-windows\sh-superh-elf\examples\os21\mandelbrot/ma

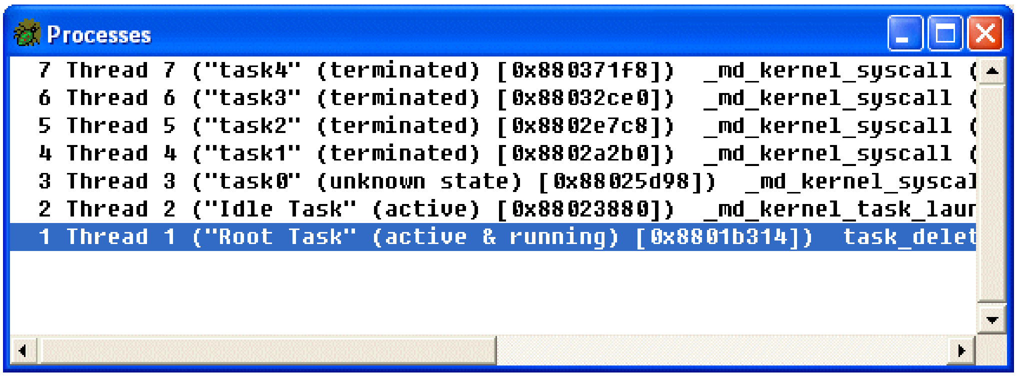
```
/*
 * Divvy up the area between the tasks
 */

for (i = 0, x = 0; x < tiles; x++)
{
  for (y = 0; y < tiles; y++, i++)
  {
    params[i].min_x = x * (XSIZE / tiles);
```

mandel.c:1 SOURCE

6.14 The Processes window

The **Processes** window (*Figure 21*) displays the active threads. To open the **Processes** window, go to the **Source Window** and from the **View** menu, select **Thread List**.

Figure 21. Processes window



```
 7 Thread 7 ("task4" (terminated) [0x880371f8])  _md_kernel_syscall (
 6 Thread 6 ("task3" (terminated) [0x88032ce0])  _md_kernel_syscall (
 5 Thread 5 ("task2" (terminated) [0x8802e7c8])  _md_kernel_syscall (
 4 Thread 4 ("task1" (terminated) [0x8802a2b0])  _md_kernel_syscall (
 3 Thread 3 ("task0" (unknown state) [0x88025d98])  _md_kernel_syscal
 2 Thread 2 ("Idle Task" (active) [0x88023880])  _md_kernel_task_laur
 1 Thread 1 ("Root Task" (active & running) [0x8801b314])  task_delet
```

The **Processes** window shows the thread number and details of the thread, such as current status. To set a thread as the current thread, click on it. This causes the debugger to switch contexts and updates all windows.

7 Building open sources

7.1 Introduction to open sources

The ST40 Micro Toolset is based on a number of open source packages which provide the compiler tools, base run-time libraries and debug tools.

The main open source packages are as follows:

- GNU binutils
- GNU GCC
- GNU GDB/Insight
- GNU make
- newlib
- zlib

Contact your ST FAE or ST support center to obtain the sources for these packages (and the other open source packages used in the toolset).

Note: STMicroelectronics does not provide support for users wishing to build these sources, beyond this short guide.

7.2 Requirements

The open source packages shipped with the ST40 Micro Toolset, have been built for the following platforms:

- Red Hat Linux Enterprise Workstation 4 and 5
- Microsoft Windows XP, Vista and 7

Note: For all platforms, the minimum version of GCC required to build the sources is 4.0.3.

The following sections describe the environment required in order to build the open source packages for each of these platforms.

7.2.1 Linux

The Linux platform should be set up for a developer (including X11 development). The following package is also required to build the open source packages:

- GNU texinfo 4.8 or later

7.2.2 Windows

To build the open source packages for the Windows platform, the Cygwin Unix emulation environment is required. The Cygwin Unix emulation environment is described in *Section A.11: Using Cygwin on page 243*.

*Note: Although the Cygwin Unix emulation environment is required to build the packages, the resulting executables do not use the Cygwin Unix emulation environment. Instead the resulting executables are standard Win32 executables created using the **MinGW** toolkit (Minimum GNU for Windows). The **MinGW** toolkit is available as a Cygwin package and should be installed together with the other required Cygwin packages.*

By default, the installer for Cygwin installs only the core set of packages suitable for an end user environment. This core set may not be sufficient to build the open source packages and so additional packages may need to be installed. The installed packages should include the following (from the **Devel** and **Doc** categories):

- ash
- binutils
- bison
- flex
- gcc
- gcc-mingw
- make
- mingw-runtime
- texinfo

The Red Hat Cygwin Unix emulation environment is available from the Cygwin website (http://cygwin.com). The MinGW toolkit is also available as a standalone package (that is, not dependent on Cygwin) from the MinGW website (http://mingw.org).

Note: The installer for Cygwin does not automatically update the Windows `PATH` *environment variable with the locations for the Cygwin tools. The following sections assume that the Cygwin tools are available in the Windows environment and therefore the Windows* `PATH` *should be manually updated with the following Cygwin tool locations:*

- `*cygwin-install-directory*\usr\local\bin`
- `*cygwin-install-directory*\usr\bin`
- `*cygwin-install-directory*\bin`
- `*cygwin-install-directory*\usr\X11R6\bin`

Where `cygwin-install-directory` *is the location of the Cygwin installation directory.*

Some of the build scripts do not work correctly when run with the default Cygwin shell (**bash**) because text files in Windows have carriage returns at the end of lines. To overcome this, replace the file `*cygwin-install-directory*/bin/sh.exe` with a copy of `*cygwin-install-directory*/bin/ash.exe`. The **sh** tool used to interpret the build scripts is now **ash**, and not the default shell, **bash**.

Cygwin's soft links are not compatible with non-Cygwin applications, such as the ST40 Micro Toolset. Therefore the Cygwin `ln.exe` tool must either be removed or overridden to prevent soft links being used. We recommend that the following short shell script is installed as *`cygwin-install-directory`*`\bin\ln` to override the default `ln.exe` tool.

```
#!/bin/sh

[ "$1" = "-s" ] && shift

exec /bin/cp -rf "$@"
```

Note: The above script `ln` *does not overwrite* `ln.exe`*, but does take precedence over it.*

7.3 Setting up the build environment

The sources of all the open source components of the toolset are contained within a single compressed **tar** archive.

Before starting to build the packages, extract this archive to a suitable location and check that the following files are present:

`src`	The directory containing the source tree.
`st40configure`	A shell script containing the correct configuration options for the ST40 Micro Toolset.
`HOWTO-BUILD`	A text file containing similar instructions to those given here. This file also contains some background information on using the GNU build system.

Before attempting the build, set up an environment that provides the proper set of build tools and has the correct environment variables set to the proper values. These settings are host specific; the following instructions provide a guide to setting up the correct build environment for various host systems.

Linux

The standard Linux developer environment is sufficient; however, the following environment variables must be set if using a 64-bit version of the operating system.

`CC`	`gcc -m32`
`CXX`	`g++ -m32`

Windows

Under Windows, it is only possible to build the toolset from within the Cygwin environment. If Cygwin has been set up in accordance with the instructions in *Section 7.2.2 on page 94*, it has most of the proper environment already set up, but the following environment variables must be set manually:

`CC`	`i686-pc-mingw32-gcc`(1)
`CXX`	`i686-pc-mingw32-g++`(1)

1. Cygwin cross compiler for creating standard Win32 executables.

7.4 Building the packages

When the environment has been set up correctly, the process of building the tools is identical for all hosts.

Create object directory

First, create an object directory at the same location as the `src` directory. This is done with the `mkdir` command:

```
mkdir objdir
```

This directory can have any name, but `objdir` is suggested. This directory holds all the intermediate files (object files) created by the build.

Note: The object directory does not have to be in this location. If it is in a different location, however, the script `st40configure` *must be modified so that the script knows where to find the* `src` *directory.*

Configuration

Set the current working directory to the object directory and run the configuration script. This is done with the following commands:

```
cd objdir
sh ../st40configure
```

The script `st40configure` invokes the GNU `configure` script with the required parameters. Any parameters passed to `st40configure` are passed through to the GNU `configure` script. The build can be customized by adding appropriate arguments to the command line.

Use the `--help` command line argument when invoking `st40configure` to obtain a list of parameters for this script. You can obtain further options by adding the `--help` command line argument to the `configure` script found in each subdirectory of `src`. Configuration options that are passed to the top level `configure` script are also passed to all subsequently called `configure` scripts within the build tree.

The parameters for the GNU `configure` script are also documented as part of the **autoconf** documentation, available from http://www.gnu.org.

Make

When the script has completed running, the object directory contains the top-level configuration files and a `Makefile`. The toolset is now ready to be built. This is done by issuing the **make** command without any parameters.

make traverses the source tree recursively, building every source file. If the build fails (for whatever reason), restart it from the place where it left off simply by invoking the **make** command again.

make creates subdirectories within the object directory to hold the object files for each of the individual packages that make up the toolset. If required, issue the **make** command within any of the subdirectories in order to rebuild any subset of the source, as represented by the chosen subdirectory.

If part of the build tree cannot be built, the affected subdirectories may be deleted and rebuilt by running the top level makefile again. `make` recreates any missing subdirectories and rebuilds their contents.

Note: Each package under `sh-superh-elf` *is replicated for each of the targets. If one copy has to be deleted, then all copies must be deleted before this package can be rebuilt.*

The entire package can be rebuilt in its entirety by deleting the object directory and starting the build process again from the configuration step (see *Configuration on page 96*). As the build process does not write to the `src` directory, the original sources are unaffected by the build process.

It is also possible to initiate multiple builds by creating more than one object directory and running `st40configure` and `make` within each directory. This may be done in order to create builds with alternative sets of build parameters. The different builds all use the same set of sources but do not interfere with one another.

Note: If using Cygwin to build the packages in a Windows environment, use the version of ***make*** *from the Cygwin tools and not the* ***make*** *that is distributed with the ST40 Micro Toolset. This can be guaranteed by using the command* `/bin/make` *from the Cygwin shell rather than* `make`*.*

Installation

When the build is successfully completed, it must be installed. The name of the installation directory is set up by the `st40configure` script, and is named `install-`*`host`*, where *`host`* is either `linux` or `windows`. The installation directory is located in the same directory as `src` and `objdir`. To install the build, invoke the following **make** command:

```
make install
```

As long as the installation is kept intact, it is possible to move the installation directory tree to any location.

If multiple builds of the toolset have been created, be aware that these may be configured with the same installation location, which means that the most recently installed build overwrites an earlier build. To avoid this, use the command line option `--prefix=`*`build`* when running `st40configure`, where *`build`* is a path that identifies the location where the build is to be installed. If each build is configured with a different *`build`* path, the different builds can be kept separate from one another.

To complete the installation, copy the non-open source components of the toolset from the official release to the equivalent location in the installation tree of the newly built toolset. For example, the toolset cannot produce executables for a silicon target without the library `libdtf.a` from the ST40 Micro Toolset.

8 Core performance analysis guide

8.1 Introduction to core performance analysis

This chapter describes how to analyze the performance of the ST40 cores using the ST40 simulator. It includes details of how to execute code on the ST40 simulator and produce statistical and trace information from these execution runs. The tools which perform the analysis of the generated data are also described.

GDB provides access to two versions of the ST40 simulator.

- The Functional Simulator provides a simulation of the functionality of the core including full instruction set simulation, memory management (MMU), and system architecture features such as caches.
- The Performance Simulator provides a simulation of the full functionality of the core. In addition, it provides cycle-accurate[a] performance information including instruction latencies, pipeline stalls, and cache behavior. This can be used to generate accurate performance trace and/or statistical information for use by the performance visualization tools.

8.2 Running performance models under GDB

This section provides examples of how to execute programs on the simulators, generate performance data and use the analysis tools.

8.2.1 Example source code

The examples in this chapter use the following program (`velocity.c`).

```
#include <stdio.h>
#include <math.h>

/* functions using basic equations of motion */
int distance(int u, int a, int t)
{
  /* s= ut + 1/2at^2 */
  int inter1, inter2;
  inter1 = u*t;
  inter2 = 0.5 * (a * pow((double)t,2.0));
  return inter1 + inter2;
}
```

[a] The performance simulator is not guaranteed to be 100% accurate in all cases.

```
int velocity(int u, int a, int t)
{
  /* v = u + at */
  return ( u + (a * t));
}

float velocity2(int u, int a, int s)
{
  /*  v^2 = u^2 + 2as */
  float inter1, inter2;
  inter1 = pow((double)u,2.0);
  inter2 = 2 * a * s;
  return sqrt(inter1 + inter2);
}

int main(void)
{
  int t = 10;
  int a = 30;
  int u = 5;
  int s,v;
  float v2;

  v = velocity(u,a,t);
  s = distance(u,a,t);
  v2 = velocity2(u,a,s);

  /*  should be the same */
  printf("Velocity 1 = %d\nVelocity 2 = %f\n",v,v2);

  return 0;
}
```

This can be compiled using the command line:

```
sh4gcc -o velocity.out -mboard=mb411sim velocity.c -lm
```

For full debugging information to be included, use the `-g` option:

```
sh4gcc -g -o velocity.out -mboard=mb411sim velocity.c -lm
```

8.2.2 Beginning a debug session

To begin a debug session invoke GDB with the command line:

```
sh4gdb velocity.out
```

It is then necessary to connect to a simulator target. For example, to use the simulator configured as an STb7100-MBoard, enter:

```
mb411psim
```

Next, load the code to be executed using the command:

```
load
```

With the code loaded into memory and the start address set, GDB leaves the simulator in a suspended mode.

To run the program use the `continue` command. This may be abbreviated to `c`.

8.2.3 Obtaining performance data

GDB creates a flexible platform from which to collect performance information. Performance logging commands can be controlled from within GDB, therefore it is possible to set breakpoints or watchpoints to turn the profiling information on or off at specific points.

Getting a trace

The following example generates a performance trace for the `velocity2()` function (see the example in *Section 8.2.1*). This requires passing commands to the ST40 simulation models using the following command:

```
sim_trace option
```

where *`option`* can be `on`, `off`, `open` or `close`.

The following example GDB command script loads the executable, opens a trace file and saves the trace information to the file specified.

```
mb411psim
load
break velocity2
continue
break *$pr
sim_trace open:velocity
sim_trace on
continue
sim_trace off
continue
quit
```

Note: The command `break *$pr` *sets a breakpoint on the PR register which contains the Program Counter (PC) value for the return of the function, effectively setting a breakpoint at the end of the function.*

The **trcview** tool is used to view the generated trace information and is described in *Section 8.5: The trace viewer (trcview) on page 120*.

To produce an instruction trace other than by using `sim_trace`, use the `sim_insttrace` command. See *sim_insttrace command on page 109* for more information.

Collecting census information

The method for collecting the census information is similar to that for obtaining trace information: `sim_census` is used instead of the `sim_trace` command. The following example GDB command script illustrates census information being collected for the `distance` function:

```
mb411psim
load
break distance
continue
break *$pr
sim_census open:velocity
sim_census on
continue
sim_census off
sim_census save:results
continue
quit
```

The **censpect** tool is used to inspect the generated census information and is described in *Section 8.4: The census inspector (censpect) on page 110*.

Using software controls

The following example shows how function calls can be used to dynamically control the collection of statistical and trace data. It is based on changing the `main()` function of the `velocity.c` example (see *Section 8.2 on page 98*) to the version below. The set of ST40 simulator dynamic control functions is described in *Section 8.3.5: Dynamic control on page 109*.

```
int main(void)
{
  int t = 10;
  int a = 30;
  int u = 5;
  int s,v;
  float v2;


  TracesOn();
  CensusOn();


  v = velocity(u,a,t);
  s = distance(u,a,t);
  v2 = velocity2(u,a,s);

  CensusOff();
  TracesOff();
  CensusOutput("Velocity statistics");

  done();
  return 0;
}
```

The example also needs to #include the include file census.h (which provides declarations for the dynamic control functions) and to be linked with the compiled source file census.c (which defines these functions). These files are located in the sh-superh-elf/examples/census subdirectory of the release installation directory.

For these dynamic control functions to have an effect, trace and census files need to be opened.

The following GDB command script example shows the velocity example being run with the version of main to turn the census on and off dynamically:

```
mb411psim
load
sim_census open:velocity
continue
quit
```

Using delayed memory models

The ST40 performance simulators support a delayed memory model. This allows the user to specify the number of cycles taken for an external memory request to be serviced by the memory system. With the model disabled, it is assumed that all memory requests are serviced instantly. It should be noted that it is not possible to use this model in conjunction with an uncached model to get “perfect” caching results. This is because the pipeline resources needed to make external accesses are still occupied resulting, for example, in contention between the instruction and data requests for the external bus.

An additional argument can be supplied when connecting to the performance simulators to enable the delayed memory models, for example:

```
mb411psim "+DMM 6"
```

instantiates the delayed memory with a latency of 6 cycles. It is also possible to alter the number of cycles delay by using the GDB command sim_command, for example:

```
sim_command "config +DMM.delay=cycles"
sim_reset
mb411stb7100sim_setup
```

Use of this method requires some care. To confirm that a sequence of configuration commands have taken effect, a sim_reset is required. More information on sim_reset is provided in *config subcommands on page 106*.

The sim_command command above sets the latency but it can also set:

- DMM.bus_width, the bus-width in bytes (default is 8)
- DMM.throughput, the through-put, in the form of the number of cycles of delay per bus-width word (the default is 1)

However, it is best that these are left at their default values.

Perfect caching

Models with caches enabled support a configuration option that causes the cache to behave as if all accesses hit the cache. It should be noted that this is the only method of collecting performance data that eliminates any influence of the cache. For example, using a zero memory model in conjunction with a cache model does not stop the pipeline interlocks and stalls that are caused by the occurrence of cache misses.

As stated in *config subcommands on page 106*, the `sim_reset` command must be issued after all configuration changes. The GDB commands to enable perfect caching (for both I-cache, and D-cache) are as follows:

```
sim_command "config +cpu.icache.perfect=true"
sim_command "config +cpu.dcache.perfect=true"
```

Setting up custom targets

To ease the configuration of the ST40 simulators, a set of custom targets can be created that contain options for setting-up the caches or enabling census taking.

The following example defines a target that opens files for census and tracing. It also sets up the model to use a 30 cycle delayed memory model. It should be noted that the method for setting-up the delayed memory model is different to the other model configuration options.

```
define mb411psim_dmm_cns
  mb411psim "+DMM 30"
  sim_census open:census
  sim_trace  open:trace
  sim_census autosave:census
end
```

The following example defines a target that sets up a model that collects tracing information and uses perfect caches. A `sim_command config` command is used and therefore must be followed by a `sim_reset` command (as described in *config subcommands on page 106*) plus any target configuration commands that have been run prior to the `sim_reset` command (in this example, `mb411stb7100sim_setup`).

```
define mb411psim_perfectc_trc
  mb411psim
  sim_command "config +cpu.icache.perfect=true"
  sim_command "config +cpu.dcache.perfect=true"
  sim_reset
  delete mem
  mb411stb7100sim_setup
  sim_trace open:trace
end
```

8.3 ST40 simulator reference

This section provides a reference for all the GDB features specific to ST40 simulator targets.

8.3.1 ST40 simulator targets

A list of all the available simulator targets can be found in the *ST40 Micro Toolset GDB command scripts user manual* (8045872).

8.3.2 shsimcmds.cmd

This command script defines the commands to control the configuration and run-time options of the ST40 simulator shipped with the ST40 Micro Toolset. The commands are listed below.

`sim_addmemory`	Define a memory region for the simulator (see *sim_addmemory command on page 108*).
`sim_branchtrace`	Enable/disable branch instruction tracing (see *sim_branchtrace command on page 108*).
`sim_census`	Enable/disable census information (see *census subcommands on page 105*).
`sim_command`	Send a generic command to the simulator (see *Section 8.3.3: ST40 simulator control commands on page 104*).
`sim_config`	Send a configuration command to the simulator (see *config subcommands on page 106*).
`sim_insttrace`	Enable/disable instruction tracing (see *sim_insttrace command on page 109*).
`sim_reset`	Resets the internal state of the ST40 simulator. For correct operation, it is required that this command is executed after every configuration change.
`sim_trace`	Enable/disable trace information (see *trace subcommands on page 107*).

8.3.3 ST40 simulator control commands

The ST40 simulator control commands are invoked using `sim_command` and must be enclosed in double quotes, for example:

```
sim_command "census open 'census'"
```

Several control commands can be specified in the same `sim_command` command using a space to separate the commands. For example, to open a census file and turn on census taking:

```
sim_command "census open 'census' census on"
```

As an abbreviation, subcommands can be combined using curly braces. The previous command is therefore equivalent to:

```
sim_command "census { open 'census' on }"
```

The details of these commands are described in the following sections.

census subcommands

Use the `census` subcommands to produce files containing statistical data that are visualized using the **censpect** tool. See *Section 8.4: The census inspector (censpect) on page 110*.

Table 21. Census subcommands

Command	Description
`open '`*`file`*`'`	Open a new census file unless a census file is already open. The *`file`* argument is the name of the census file to be created (without the `.cns` extension which is added automatically) and must be enclosed by single quotes. An alternative for this command is `sim_census open:`*`file`*. Census taking does not commence until a `census on` command.
`on`	Switch on census taking. Subsequently executed instructions contribute to the statistical data being collected until an `off` command or the program terminates. An alternative for this command is `sim_census on`.
`off`	Switch off census taking. Subsequently executed instructions do not contribute to the statistical data being collected until an `on` command. It is not essential to switch off census taking before issuing a `save` command. An alternative for this command is `sim_census off`.
`reset`	Reset the census counters. Any census information not already saved is lost. This is required after a `config` command. An alternative for this command is `sim_census reset`.
`save '`*`label`*`'`	Save the current state of the census counters to the census file. *`label`* specifies the label for the census record that is generated and must be enclosed by single quotes. An alternative for this command is `sim_census save:`*`label`*.
`autosave '`*`label`*`'`	Set up a delayed save that is performed when the current census file is closed. *`label`* specifies the label for the census record that is generated and must be enclosed by single quotes. An alternative for this command is `sim_census autosave:`*`label`*.

config subcommands

Use the `config` subcommands to review or modify the configuration variables of the current ST40 simulation.

A `sim_reset` must be performed after any sequence of these commands in order for changes to take effect. This has the side-effect of performing a hard-reset of the core: an operation which modifies the contents of some registers, for example the CCR (cache control register), which may have been initialized during the target setup. Therefore it is necessary to follow a `sim_reset` command with a re-initialization of the target, typically using a `target_setup` command (see *Setting up custom targets on page 103* for an example).

Table 22. Config subcommands

Command	Description
`+variable=value`	Modify a configuration variable associated with the model. The variables for both the functional simulator and the performance simulator are listed in *Table 72 on page 284*. Additional variables for the performance simulator only are listed in *Table 73 on page 285*. The current values of the configuration variables can be determined using the `save` command or by reviewing the `SIM.CONFIG` section of a generated census file.
`load 'file'`	Load the configuration variables stored in `file.cfg`. The format of the file is described in *Section 8.7: Census file formats on page 124*. `file` must be enclosed by single quotes. It is possible to load several configuration files. If any variable is defined more than once then the last value specified for a variable is used.
`save 'file'`	Save the current value of the configuration variables in configuration file format (see *Section 8.7: Census file formats on page 124*) to `file.cfg`. The saved state of the configuration variables can therefore be restored at anytime by reloading `file`. `file` must be enclosed by single quotes.

A complete list of all configuration variables for the ST40 simulator can be found in *Appendix G: Simulator configuration variables on page 284*.

trace subcommands

Use the `trace` subcommands to produce files containing statistical data that is then visualized using the **trcview** tool described in *Section 8.5: The trace viewer (trcview) on page 120*.

Table 23. Trace subcommands

Command	Description
`open 'file'`	Open a set of trace files with the base name `file`. Depending upon the type of simulation being traced, a number of files are created and various extensions added to the trace base name. See the description of the trace file format in *Section 8.6: Trace viewer file formats on page 122* for more details. `file` must be enclosed by single quotes. An alternative for this command is `sim_trace open:file`. Tracing does not commence until a `trace on` command.
`on`	Switch on tracing of the simulation. An alternative for this command is `sim_trace on`.
`off`	Switch off tracing of the simulation. An alternative for this command is `sim_trace off`.
`close`	Flush and close all the files associated with the current trace. An alternative for this command is `sim_trace close`.

Table 24 lists the equivalent `sim_command` commands for the `census`, `config` and `trace` commands listed in *Table 21*, *Table 22* and *Table 23*.

Table 24. sim_command equivalents

Command	sim_command equivalent command
`sim_census on|off`	`sim_command "census on|off"`
`sim_census save:label`	`sim_command "census save 'label'"`
`sim_census autosave:label`	`sim_command "census autosave 'label'"`
`sim_census open:file`	`sim_command "census open 'file'"`
`sim_config "config"`	`sim_command "config +config"`
`sim_trace on|off`	`sim_command "trace on|off"`
`sim_trace open:file`	`sim_command "trace open 'file'"`
`sim_trace close`	`sim_command "trace close"`

Note: The `census` and `trace` commands are only available when using the ST40 performance simulator.

8.3.4 Commands in shsimcmds.cmd

The command script `shsimcmds.cmd` also defines the simulator commands `sim_addmemory`, `sim_branchtrace` and `sim_insttrace`. These commands are described here.

sim_addmemory command

Use the `sim_addmemory` command to add a memory region to the simulator. This command accepts three arguments:

```
sim_addmemory address size type
```

The arguments are as follows:

address	The address of the memory region, in hexadecimal.
size	The size of the memory region in Mbytes.
type	One of either `RAM`, `ROM` or `DEV`.

sim_branchtrace command

Use the `sim_branchtrace` command to configure branch tracing in the simulator. The command has the format:

```
sim_branchtrace option
```

Table 25 describes the options that can be used with `sim_branchtrace`.

Table 25. branchtrace subcommands

Option	Description
`open:`*`file`*	Record the branch trace data in *`file`*. If the `open` command is not used, then the simulator uses a file with the default name of `brtrace.dat`. If *`file`* already exists, new branch trace records are appended to the end of the file. Branch tracing does not commence until a `branchtrace on` command is issued. If branch trace is already enabled, the file can be changed dynamically without the need to disable and then re-enable branch tracing.
`on`	Turn branch tracing on.
`off`	Turn branch tracing off.

The following commands open a file called `velocity.dat` and start branch tracing:

```
sim_branchtrace open:velocity.dat
sim_branchtrace on
```

Each branch is recorded in the output file using the following simple format:

```
source-address -> destination-address
```

sim_insttrace command

Use the `sim_insttrace` command to enable or disable the listing of executed instructions within the simulation. The list of instructions are displayed on `stderr` in the terminal from which GDB was launched.

The command to enable instruction trace is as follows:

```
sim_insttrace on
```

The command to disable instruction trace is as follows:

```
sim_insttrace off
```

8.3.5 Dynamic control

Dynamic control over the collection of statistical and trace data is performed through a pseudo-device mapped into the address space of the core. The code needed to drive this device is encapsulated within C functions that are implemented in `census.c` located in the `sh-superh-elf/examples/census` subdirectory of the release installation directory.

Note: The simulation control is not overly intrusive. Most commands involve executing relatively few instructions and, in most cases, a single write to the dynamic control device. However, the `CensusOutput()` *function involves the execution of significant code, particularly when a string is copied into the dynamic control device. It is therefore highly recommended that the* `CensusOff()` *function is invoked prior to calling the* `CensusOutput()` *function. This avoids the execution of the* `CensusOutput()` *function appearing in the statistics.*

The associated header file `census.h` defines the functions listed below.

GetClock Return the current clock cycle

Definition: `int GetClock(void)`

Description: Return the current clock cycle during a performance simulation and an instruction count during a functional simulation.

CensusOn Switch on census taking

Definition: void CensusOn(void)

Description: Switch on census taking. It is equivalent to setting a breakpoint at that particular point in the code and executing a `census on` command.

CensusOff Switch off census taking

Definition: `void CensusOff(void)`

Description: Switch off census taking. It is equivalent to setting a breakpoint at that particular point in the code and executing a `census off` command.

CensusClear Reset the census counters

Definition: `void CensusClear(void)`

Description: Reset the census counters. It is equivalent to setting a breakpoint at that particular point in the code and executing a `census reset` command.

CensusOutput Save the census counters

Definition: `void CensusOutput(char* label)`

Description: Save the census counters to the `census` file with the label `label`.

TracesOn Enable tracing

Description: Enable tracing. It is equivalent to setting a breakpoint at that particular point in the code and executing a `trace on` command.

Definition: `void TracesOn(void)`

TracesOff Disable tracing

Description: Disable tracing. It is equivalent to setting a breakpoint at that particular point in the code and executing a `trace off` command.

Definition: `void TracesOff(void)`

8.4 The census inspector (censpect)

The census inspector provides a means of visualizing the contents of census files. This section describes it in the context of the census information produced by the ST40 simulators, but it can be used to view the content of any census file that adheres to the format described in *Section 8.7: Census file formats on page 124*.

The tool is called **censpect** and takes a single optional argument which is the name of the configuration file into which configuration data can be loaded and saved. If no argument is specified then the default configuration file is used, `.censpect.v3.cfg`, which is found either in the user’s home directory under Linux or in the root of the C: drive under Windows.

If this is not present, then a configuration file is created based on the default internal configuration. To select the default configuration explicitly, use `-` (hyphen) as the argument:

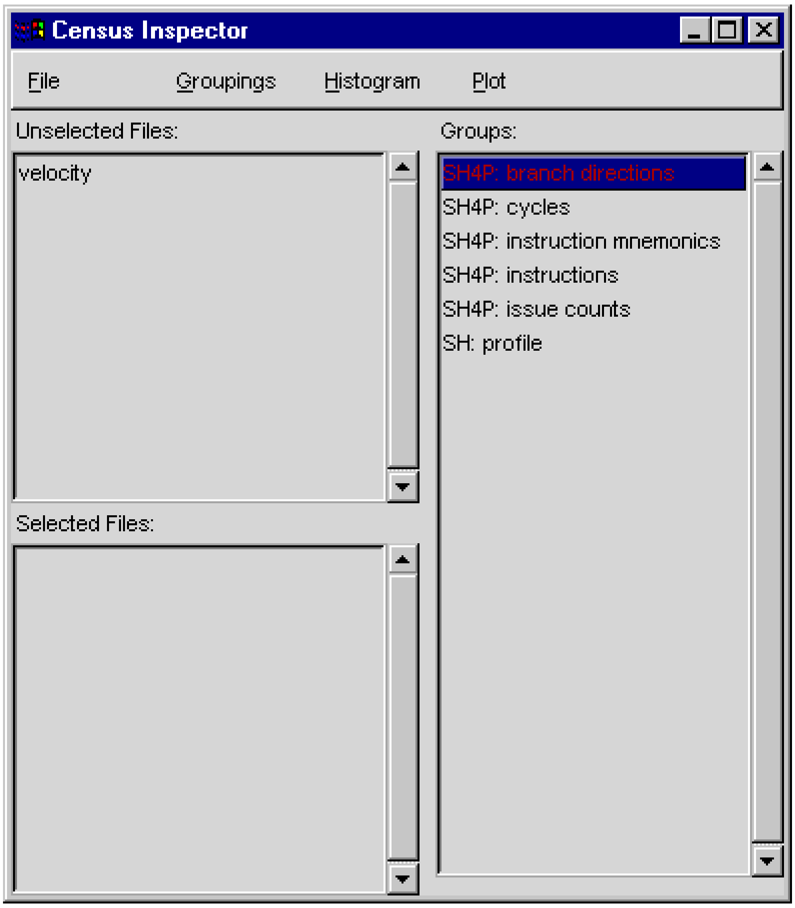
```
censpect -
```

If this argument is used, it is not possible to save any configuration changes.

8.4.1 The Census Inspector window

An example of the **Census Inspector** window is shown in *Figure 22*.

Figure 22. Census Inspector window



The fields listed in *Table 26* are available in the **Census Inspector** window.

Table 26. Fields available in the Census Inspector window

Field	Description
Unselected Files	This contains a list of all the census files that the tool has located in the current directory. To load a file into the tool, click on the appropriate entry. Several files can be loaded at the same time by selecting multiple entries when clicking.
Selected Files	Loaded files are transferred to **Selected Files** and prefixed by the name of the tool or model by which they were generated. To unload a file, click on the appropriate entry.
Groups	Related queries can be assembled into groups whose results are displayed as histograms or 2D plots as appropriate. A number of useful groups are pre-defined and are listed in **Groups**. Generally, only the pre-defined groups with a prefix matching the model/tool type specified in the loaded census file are appropriate. It is recommended that this convention is adopted when adding user defined groupings. The creation of new groups is described in *Section 8.4.4 on page 116*.

To make all census files located below the current directory available for selection, select **Search Subdirectories** from the **File** menu. Selecting the **Follow Links** check box causes the search to traverse any symbolic links that it encounters (not applicable to Windows).

To change the working directory of the tool, select **Change Directory** from the **File** menu.

Any new census files generated while the tool is running do not automatically appear in the **Unselected Files** list. To update this window select **Rescan Directory** from the **File** menu.

8.4.2 Creating histograms

Once a group has been defined and a suitable census file loaded, it is then possible to create a histogram. To do this, double-click on the appropriate entry in the **Group** list. Alternatively select the group with a single click and then select **Display Histogram** from the **Histogram** menu.

The **Display Histogram** function applies each query found in the selected group to the census database and builds a table of queries and associated counts. These results are then displayed on the screen in the form of a bar chart (or histogram).

The pre-defined groups supplied with the tool typically aggregate across all the census records[(b)] found in the loaded census files. To display specific records from the loaded census files, select **Select Output** from the **Histogram** menu.

The **Select Output** function lists all the census records in the census database by output number and label. Arbitrary numbers of the records can then be selected using the same mechanism as that used for loading files. The results contained within the chosen records can then be merged (by summing the results of applying the same query to each selected record) or displayed separately. If only one label is selected these two options have exactly the same effect.

[b] A census record is defined as the data produced by a simulator each time the `CensusOutput()` function is called.

A number of additional options are also available on the **Histogram** menu (see *Table 27*). These control the way in which the histogram is formatted on the screen.

Table 27. Histogram menu options

Menu option	Description
Use Blt Package	Draw the bar charts using a different histogram package. The advantages of this package are that it is capable of marking the ruler with percentages instead of counts, and drawing the labels vertically. This allows more columns to be placed onto the screen. The disadvantages are that bars can only be displayed vertically, charts can be no wider than the screen and there is no cut-off facility so that short bars can be omitted from the display.
Show Zero Bars	If this option is set then a bar continues to be displayed for any queries that return zero.
Show Count	Display the counts returned by the queries as an annotation to each bar.
Show Files	Display the file names of the census files used to produce the histogram at the top of the window.
Show Query	The histogram viewer takes a table containing the results of queries applied to the database as its input. It determines the label that appears next to each bar by stripping any prefix and any suffix common to all the queries in the table. This option displays these common components of the queries in the form: `prefixXsuffix`
Show Percentage	Annotate each bar with a percentage. Whether this is a percentage of a summation applied across all the columns or of the highest count is determined by the **Percentage of Sum** and **Percentage of Max** menu entries.
Unsorted	Display the bars in the same order which they appeared in the group.
Sort Labels	Order the bars by breaking up the labels into alphabetic and numeric sequences. The labels are then sorted from left to right using dictionary-order for alphabetic strings and value-order for numeric strings.
Sort Increasing	Organize the bars with the shortest first.
Sort Decreasing	Organize the bars with the tallest first.
Y is Count	Mark the ruler with counts.
Y is % of Sum	Mark the ruler with a percentage of the column sum.
Y is % of Max	Mark the ruler with a percentage of the tallest column.
Y is % of Group	Allow the percentage to be expressed as a count produced when applying a specified group to the census database.
Horizontal Bars	Display the bars of the histogram horizontally instead of vertically. This option is not available when **Use Blt Package** is in use.

Table 27. Histogram menu options (continued)

Menu option	Description
Wide Results	If this function is selected and the census query returns more results than can be presented on the screen, a scroll bar is automatically added to the display. This can then be used to navigate to the appropriate part of the chart. If it is not selected, then the counts associated with bars that cannot be displayed are summed and placed in a bar labelled **REST**. This option is not available when **Use Blt Package** is in use.
Select Cutoff	Display the **Set Cutoff Point** window. This window can be used to select the cutoff point by: – clicking on **None** to disable the cutoff mechanism, – clicking on **Accumulate into OTHERS** to add together all columns which contribute less than the cutoff percentage to the column sum and display it in a bar labelled **OTHERS**, – clicking on **Forget OTHERS** to ignore all columns which contribute less than the cutoff percentage, – entering the percentage of the maximum to use as the cutoff point in the **Enter cutoff** field. This facility is not available when **Use Blt Package** is in use.

8.4.3 2D plots

The **censpect** tool supports two modes of graph generation which are selected from the **Plot** menu. These can be based on:

- census record labels
- changes to the simulator configuration

Plots based on census record labels

If the **X taken from label** option is selected, plots are produced that are dependent upon the contents of the labels associated with each census record. The X co-ordinate of a point is taken from the label of the record. The Y co-ordinate is produced by using the selected group to query the census record.

Each member of the group is expected to return a single count value[c] that is used to locate the Y co-ordinate for the line. A line is plotted on the graph for every query that appears in the group. When this option is selected, a list of merged labels is supplied. Labels are merged when they differ by only a single token and the differing tokens are numeric.

The result of a merge is to produce a single label of the form:

```
prefixXsuffix (for X = token-list)
```

[c] This is achieved by giving the full tag name of a census item and only using a wild card for the `CENSUS.OUTPUTn` field; for example `CENSUS.+`. See *Census queries on page 117* for more information on queries.

For example, the following set of labels:

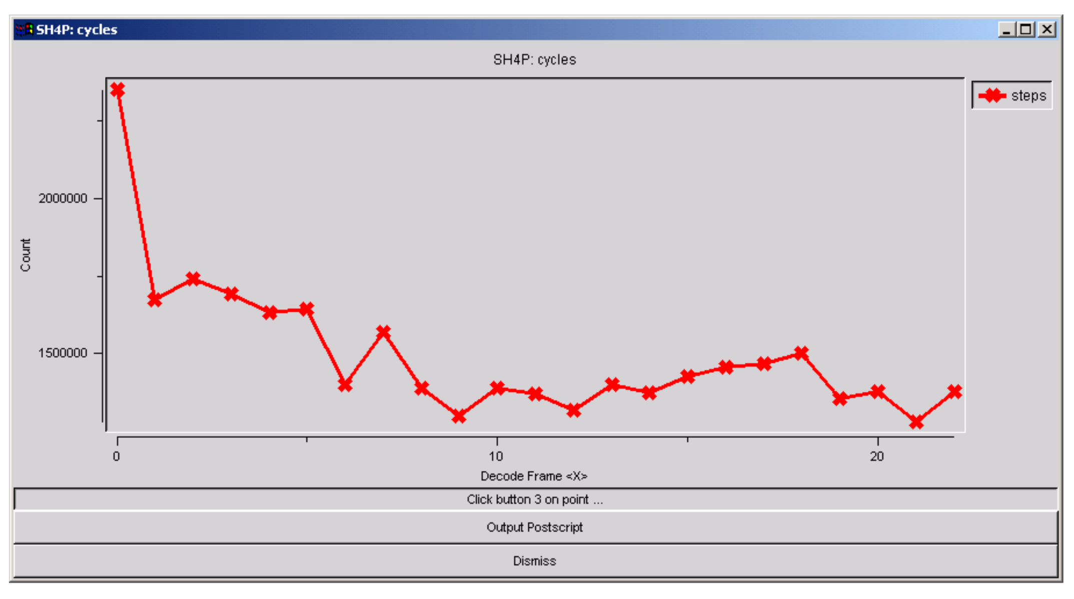
```
Decode Frame 0
Decode Frame 1
Decode Frame 2
Decode Frame 3
```

would be processed into the following merged list:

```
Decode Frame X (for X = 0 1 2 3 4 ...)
```

The required plot is selected by double-clicking on the appropriate entry in the list. The sample plot shown in *Figure 23* displays the range of cycles required to compress a sequence of MPEG frames.

Figure 23. Cycles required to compress a sequence of MPEG frames



Plots based on changes to the simulator configuration

This plotting function is used when the **X taken from configuration** option is selected. It graphically represents the effect of varying the configuration of the simulator on the execution of an application. This function is provided primarily for observing the effects of changes in the architecture and micro-architecture. For example, it is possible to observe the effect of varying the cache size on execution times and miss rates.

Before using this function, it is necessary to generate a set of census files that run the same application but use different simulator configurations.

Once a set of census files have been generated they should be loaded into the tool and **X taken from configuration** selected. This interrogates the configuration information recorded in each of the census files to determine which configuration parameter is varying. A plot is not produced if more than one parameter is found to be changing across the set of census files.

When the varying parameter has been identified, a line is drawn for every result returned by applying the currently selected census group to one of the selected census files. The X co-ordinate identifies the value of the configuration parameter. The Y co-ordinate is the count returned by the associated query.

Plotting options

The options listed in *Table 28* are also available from the **Plot** menu.

Table 28. Plot menu options

Menu option	Description
Unmarked	Do not mark the points that are used to draw the plot.
Circular	Mark the points that are used to draw the plot as circles.
Square	Mark the points that are used to draw the plot as squares.
Diamond	Mark the points that are used to draw the plot as diamonds.
Cross	Mark the points that are used to draw the plot as crosses.
Y is Count	Mark the Y axis ruler in terms of counts.
Y is % of Group	Mark the Y axis ruler in terms of a percentage of a group specified value. The group used to compute this value is chosen using **Select Base Group** on the **Groupings** menu.

8.4.4 Preparing new groups

The preparation of new groups requires an understanding of the format and content of census files and also knowledge of how to query the census database.

Census file format

This section discusses the generic format of census files. Detailed information regarding the content of census files generated by the ST40 simulators can be found in *Section 8.7: Census file formats on page 124*.

A census file entry consists of a tag/value pair where the value can be either an integer or a quoted string of characters. For example:

```
SIM.TYPE "SH4P"
SIM.DATA.CACHE.BYPASS 0
SIM.DATA.CACHE.SETS 64
SIM.CODE.CACHE.BYPASS 0
SIM.CODE.CACHE.SETS 64
```

The order in which entries appear in the census file is of no consequence to their interpretation, although to improve clarity, related information should be grouped together.

Tag names must start with a character from the set:

```
_ $ # % A-Z a-z
```

In the body of the tag, plus (`+`), minus (`-`) and numerals are also permitted. The census database is case-insensitive. Therefore, `SIM`, `Sim` and `sim` are all equivalent.

Census queries

The least complex form of a census query is a dot-separated list of field names. The query returns every entry that is prefixed by a matching set of field names. Field matching is case-insensitive. This means that the query `sim.data` would return the following entries if applied to the above census database:

```
SIM.DATA.CACHE.BYPASS 0
SIM.DATA.CACHE.SETS 64
```

In addition to prefix matching, there are two operators defined for querying the database that can be used to wildcard a particular field. If the `?` operator is used as a field name, the matching operation on that field is the one that always succeeds. `sim.?.cache.sets` returns:

```
SIM.DATA.CACHE.SETS 64
SIM.CODE.CACHE.SETS 64
```

The `+` operator is used in a similar manner except that the counts returned by the matching entries are summed to return a single result. `sim.+.cache.sets` returns:

```
SIM.+.CACHE.SETS 128
```

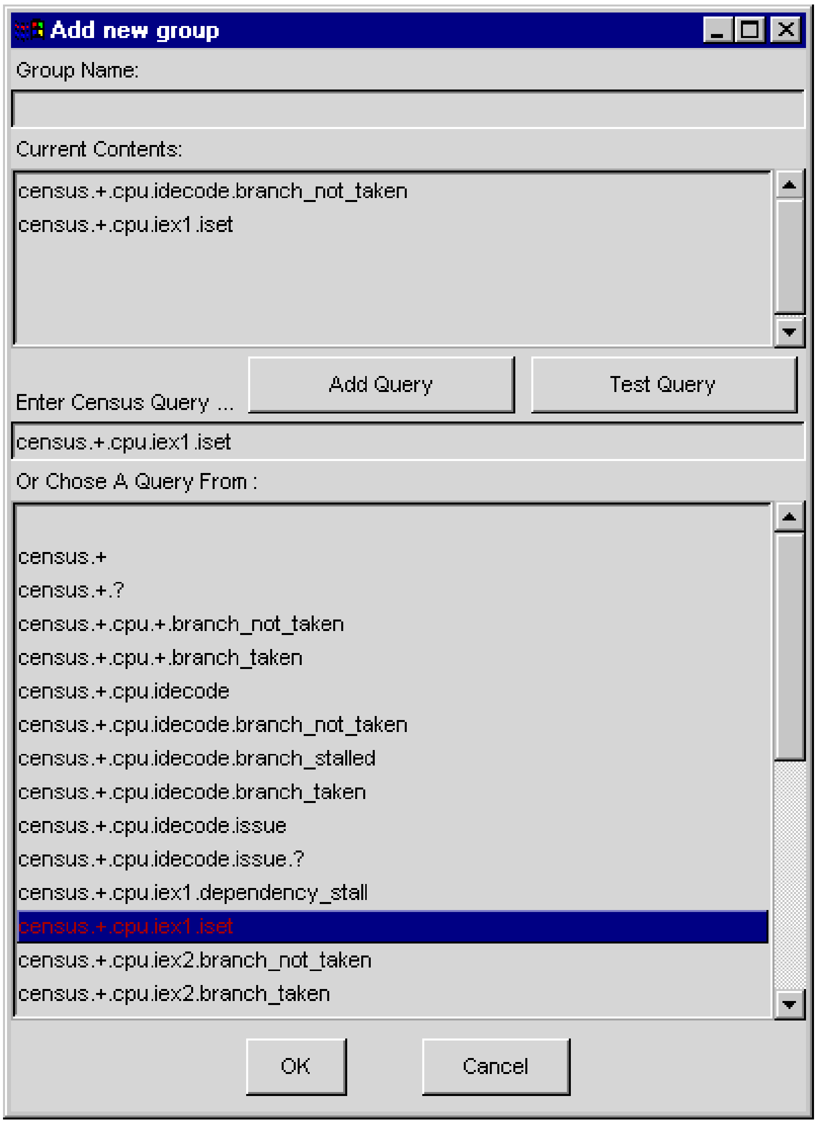
Both operators can appear multiple times in any combination within the same query. For example, `sim.+.cache.?` returns:

```
SIM.+.CACHE.BYPASS 0
SIM.+.CACHE.SETS 128
```

8.4.5 Creating and modifying groups

To begin creating a new group, select **New Group** from the **Groupings** menu. The **Add new group** window is displayed, see *Figure 24*. The fields are described in *Figure 29*.

Figure 24. Add new group window



Group Name:

Current Contents:

census.+.cpu.idecode.branch_not_taken
census.+.cpu.iex1.iset

Enter Census Query ...

census.+.cpu.iex1.iset

Or Chose A Query From :

census.+
census.+.?
census.+.cpu.+.branch_not_taken
census.+.cpu.+.branch_taken
census.+.cpu.idecode
census.+.cpu.idecode.branch_not_taken
census.+.cpu.idecode.branch_stalled
census.+.cpu.idecode.branch_taken
census.+.cpu.idecode.issue
census.+.cpu.idecode.issue.?
census.+.cpu.iex1.dependency_stall
census.+.cpu.iex1.iset
census.+.cpu.iex2.branch_not_taken
census.+.cpu.iex2.branch_taken

Table 29. Add new workgroup window fields

Field	Description
Group Name	Enter the name of the group. The recommended convention is to prefix the group name with the type information for the tool/model that produces the census files with which it is associated. This information can be found in the census file's `SIM.TYPE` field and prefixes a selected census name.
Current Contents	This lists all of the queries in the group. To delete a query from the group double-click on the appropriate entry.
Enter Census Query	Enter the query to add to the group.
Add Query	Click on this button to add the query entered in the **Enter Census Query** field to the group. The query is applied to the currently loaded census files and a table of results is displayed in a new window. Any query which has no match against the loaded census files returns `0`. This mechanism is also advantageous for inspecting the contents of a census file. For example, testing `census.+.cpu0` lists all census file entries associated with CPU core 0. Most queries should commence `census.+` as this makes the default behavior of the query merge the results of every census record found in the database. This behavior can be overridden using the **Select Output** option in the **Histogram** menu to specify arbitrary subsets of the census records for display.
Test Query	Click on this button to test the query entered in the **Enter Census Query** field before adding it to the group. When constructing queries for processing by the census inspector, only census file entries with numeric values are appropriate. Attempting to use queries which return strings as part of a query grouping result in an error.
Or Choose A Query From	This lists a history of all previously used queries[1]. Click on an entry in this box to transfer it to the **Enter Census Query** field. Double-click on an entry to transfer it to the **Current Contents** field. The history list also contains queries of the form `merge(`*`group-name`*`)` for each of the groups that are currently defined. A query of this form applies the query group *`group-name`* to the census database. The results of this query are accumulated and returned by the merge query as a single result.

1. To edit the contents of the history list, select **File > Selector Maintenance**.

To save the new group of queries, click on the **OK** button.

To delete a group, select it in the **Census Inspector** window and then select **Delete Group** from the **Groupings** menu. **Modify Group** displays the same dialog box as used for adding a new group, pre-loaded with the contents of the currently selected group. This window can also be opened by double-clicking on an entry in the **Groups** list.

8.5 The trace viewer (trcview)

The trace viewer is invoked as follows:

```
trcview file[.trc]
```

where *file* is the file name for the trace set to be viewed.

When the trace viewer is invoked, the **Trace File Viewer** window is displayed, see *Figure 25*.

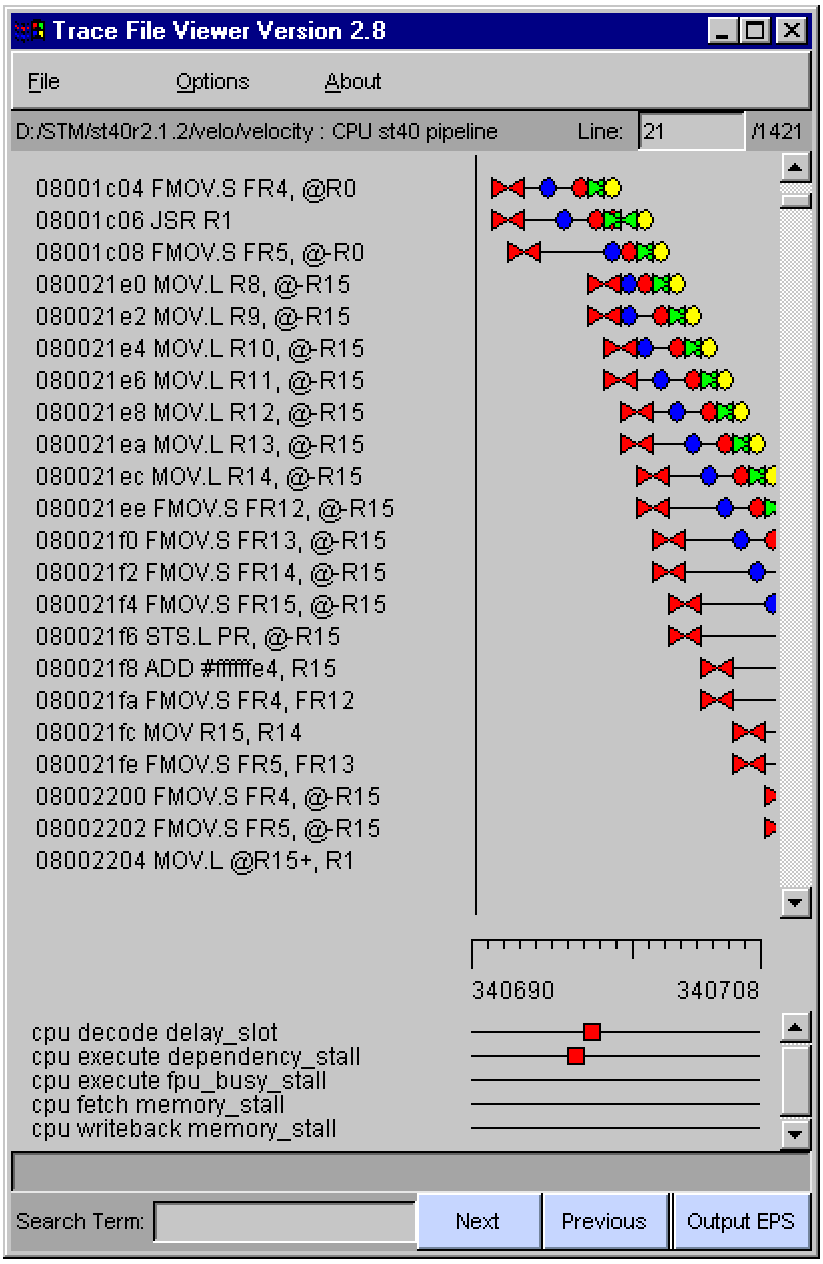
The **Trace File Viewer** window is able to display the following:

- packet traces
- probe traces

Packet traces are used for time-stamping the flow of an instruction through a core pipeline.

All information regarding the specifics of the trace are encapsulated within the generated trace files themselves. The number of packet traces generated depends upon the model. A list of traces is displayed at the bottom of the **Options** menu. Traces show the flow of the instructions through each stage of its pipeline. An example of such a trace is given in *Figure 25*.

Figure 25. The Trace File Viewer window



D:/STM/st40r2.1.2/velo/velocity : CPU st40 pipeline

```
08001c04 FMOV.S FR4, @R0
08001c06 JSR R1
08001c08 FMOV.S FR5, @-R0
080021e0 MOV.L R8, @-R15
080021e2 MOV.L R9, @-R15
080021e4 MOV.L R10, @-R15
080021e6 MOV.L R11, @-R15
080021e8 MOV.L R12, @-R15
080021ea MOV.L R13, @-R15
080021ec MOV.L R14, @-R15
080021ee FMOV.S FR12, @-R15
080021f0 FMOV.S FR13, @-R15
080021f2 FMOV.S FR14, @-R15
080021f4 FMOV.S FR15, @-R15
080021f6 STS.L PR, @-R15
080021f8 ADD #ffffffe4, R15
080021fa FMOV.S FR4, FR12
080021fc MOV R15, R14
080021fe FMOV.S FR5, FR13
08002200 FMOV.S FR4, @-R15
08002202 FMOV.S FR5, @-R15
08002204 MOV.L @R15+, R1
```

```
cpu decode delay_slot
cpu execute dependency_stall
cpu execute fpu_busy_stall
cpu fetch memory_stall
cpu writeback memory_stall
```

Probe traces are used to monitor the state of individual components of the model on a cycle-by-cycle basis. All probes are boolean, for example, a probe can be used to indicate when a queue is empty. However, it cannot be used to indicate how many entries are in the queue unless a probe is allocated for every possible number of queue entries.

Probes appear at the bottom of the **Trace File Viewer** window but can be disabled by unchecking the **Probes** option in the **Options** menu.

Additional information regarding all the probes can be displayed by selecting **Probe Configure** from the **File** menu. The dialog box displayed allows probes to be hidden, to have the color changed and to be placed on the same line as the previous probe. The latter feature, in combination with a change in color, is advantageous for presenting related signals on a single line. An example of this is the treatment of EMI phases which appear, by default, on a single line with the RAS phase in red, the CAS phase in purple, and any pre-charge phase in blue.

8.6 Trace viewer file formats

This section describes the format required for the files read by the trace viewer (see *Section 8.5: The trace viewer (trcview) on page 120*).

Note: Each file which makes up a trace set has the same file name but with a different extension. The additional extensions are defined in the `.trc` file (a member of the trace set).

8.6.1 Trace set files (.trc)

The `.trc` file gives the general structure of the trace set. The types of packet trace used in the set are defined, as are the possible stamps[d] which can be used in each packet trace. The file also contains references to the files containing the values used to construct the traces.

Definitions of trace types

The following construct defines a trace packet type with an entry for each available stamp:

```
DEFINE TRACETYPE type-name
{stamp-name} shape color
{stamp-name} shape color
...
END
```

where:

`type-name` is the name used to refer to the trace type,

`stamp-name` is the name that appears in the legend,

`shape` is the default shape the stamp uses, this can be square, circle, diamond, left arrow, or right arrow,

`color` is the default color the stamp uses.

A trace type definition must exist for each referenced trace type.

[d] A stamp is an event that can occur zero or more times during the course of the trace. For instruction traces, there is one stamp for each of the various decode and execute states of the processor pipeline.

References to associated files

An entry must exist for each packet trace in the set specified by a `TRACE` statement.

`TRACE` *`type-name file-extension`* `"`*`trace-name`*`"`

where:

`type-name` is the name of the trace type (as defined above),

`file-extension` is the extension of the file containing the data for this trace,

`trace-name` is the name of the trace as it appears in the view.

Following each `TRACE` statement there can be any number of `TRACETEXT` statements that refer to files containing additional trace information (see *Section 8.6.3 on page 123*).

`TRACETEXT` *`file-extension`*

There must also exist entries for each probe trace file (see *Section 8.6.4 on page 124*), specified by a `PROBE` statement.

`PROBE` *`file-extension`*

8.6.2 Packet trace files

Packet trace files are referred to in the `TRACE` statements of a `.trc` file. They must contain a line for each object in the trace. Each line should be structured as follows:

`:` *`descriptive-text`* `(`*`values`*`)`

where:

`descriptive-text` is any arbitrary string up to the `(` character that is displayed next to the trace of the object in the viewer,

`values` is a list of the time stamps for the object where each stamp can be represented using:

- a single integer value,
- multiple values (these must be enclosed in braces, for example `{1 2 5}`),
- empty braces `{}`.

Each value (or set of braces) must be separated by a space. Using multiple values causes more than one instance of the stamp in the trace.

Note: There is a deprecated form for an entry which has a value before the colon. In the current version, everything before the colon is ignored.

8.6.3 Trace text files

Trace text files contain one line of free text for each line in the packet trace file. The trace viewer displays the text, one line at a time, at the foot of the packet trace. Typically the text contains the same information as the packet trace descriptive text, but with more information (such as register contents).

8.6.4 Probe trace files

Probe trace files contain a header plus the trace data for the probes which are displayed by the trace viewer window. The performance simulator uses these files to represent pipeline stalls.

Header

The header starts with a line containing the word PRELUDE and then follows one line for each probe type:

word bit name description

where:

word is the binary word in which the *bit* is found. The value should be 0, 1, 2 and so on.

bit is the binary bit (0, 1, 2 and so on) that represents the probe in the data (see the trace data below). It must be between 0 and 15 (inclusive). More can be used by using a different word.

name is the name displayed by the trace viewer.

description is a short description of what triggers the probe.

Trace

The trace follows the header and starts with a line containing the word TRACE and then follows a series of lines describing the probe data:

value probes ...

where:

value corresponds to the values in the packet trace data (see *Section 8.6.2 on page 123*).

probes indicates which probes were triggered. This is a hexedecimal number (without a 0x prefix) in which the binary bits represent the probes (as defined in the probe trace file header).

... is optional and means that if there are multiple words defined in the header (that is, more than 16 probes) then these should be listed, separated by spaces.

8.7 Census file formats

Every census file (.cns) generated by the ST40 simulators contains the fields shown in *Table 30*. Census files are read by the census inspector (see *Section 8.4 on page 110*).

Table 30. Census file generic fields

Field	Description
SIM.TYPE	The type of model which generated census data.
SIM.VERSION	The version of the model which generated census data.
SIM.ARCH	The model family, usually empty.
SIM.BUILD.DATE	The date the model was built.
SIM.RUN.DATE	The date the model was run.

These are followed by a series of model specific configuration fields that contain the settings for all the run-time configurable components of the model. They are all prefixed by `SIM.CONFIG`.

The configuration section is followed by a series of counter descriptions of the form:

```
DESCRIPTION.<counter id> "<description>"
```

For example:

```
DESCRIPTION.cpu.decode.branch_taken  "Count of branches predicted
to be taken by the decode pipeline stage"
```

The description section is followed by a series of output sections, one per counter dump performed during simulation. Each entry begins:

```
CENSUS.OUTPUTn
```

where *n* begins at 0 and is incremented at the start of each new output section. Each output section contains a `LABEL` field describing its contents as specified during the counter dump. The rest of the entries record the state of all the counters at the time the dump was requested.

Some of this census data is output in the form of a histogram. This type of entry always contains the subfields shown in *Table 31*.

Table 31. Census file subfields

Subfield	Description
`total`	Sum of all values entered in the histogram.
`samples`	Number of samples in the histogram.
`min`	Lowest value entered in the histogram.
`max`	Highest value entered in the histogram.
`mean`	Mean of all values entered in the histogram (truncated to an integer).
`bins.#-X`	Number of values below or equal to `X` entered in the histogram.
`bins.#X-Y`	Number of values between `X` and `Y`, inclusive, entered in the histogram.
`bins.#X`	Number of values equal to `X` entered in the histogram.
`bins.#X-`	Number of values greater than or equal to `X` entered in the histogram.

The number, widths and range of `bins` vary depending upon the data being sampled.

9 OS21 source guide

The source code for OS21 is included with the toolset release and is located in the `sh-superh-elf/src/os21` subdirectory of the release installation directory.

OS21 may be freely rebuilt for your own purposes, but these uses must be strictly within the terms and conditions of the OS21 Software License Agreement. A copy of this license agreement is located in the top level directory of the OS21 source code (`LICENSE.htm`).

There is a `makefile` (GNU make compatible) in this directory, which enables OS21 and its board support libraries to be built by simply issuing the `make` command.

This top level `makefile` has three build rules.

build Build all of OS21 and its board support libraries (the default rule).

buildbsp Build just the OS21 board support libraries.

clean Remove all built files (object files and libraries).

The resulting libraries are placed in the directory `sh-superh-elf/src/os21/lib/st40`.

The source for OS21 is provided so that a user can:

- refer to the OS21 source for a clearer understanding of OS21’s behavior
- refer to the OS21 source to aid debugging
- rebuild OS21 with different compiler options
- enable configurable options within OS21 which are not enabled in the shipped binaries
- build customized board support libraries

Note: To build OS21, GNU make and Perl (version 5.6.1 or later) must be available on the host.

9.1 Configuration options

OS21 supports a number of configuration options. These options are selectively enabled at build time by defining preprocessor symbols. *Table 32* lists the preprocessor symbols that are available for configuring OS21 for the ST40.

Table 32. OS21 configurable options

Symbol name	Description
`CONF_DEBUG`	Enable debug checking within the OS21 kernel.
`CONF_DEBUG_ALLOC`	Enable additional debug checking for memory allocators.
`CONF_DEBUG_CHECK_EVT`	Perform extra validation checks on events.
`CONF_DEBUG_CHECK_MTX`	Perform extra validation checks on mutexes.
`CONF_DEBUG_CHECK_SEM`	Perform extra validation checks on semaphores.
`CONF_FINE_GRAIN_CLOCK`	Program the system clock to operate at as high a frequency as possible, hence yielding greater accuracy.
`CONF_FPU_SINGLE_BANK`	Restrict FPU save and restore to the bank of FPU registers used by GCC.

Table 32. OS21 configurable options (continued)

Symbol name	Description
CONF_INLINE_FUNCTIONS	Inline certain functions.
CONF_NO_FPU_SUPPORT	Do not save/restore FPU registers on context switch.

These configurable options are described in detail in Section 9.1.1.

Note: The file makest40.inc *(located in the at the top level of the OS21 source code directory) can be amended to alter these options. By default, none of the configuration options listed in Table 32 are enabled in this file.*

9.1.1 Configuration options in the standard OS21 libraries

The standard OS21 libraries shipped in the distribution (selected with `-mruntime=os21`) are built with `CONF_INLINE_FUNCTIONS` defined. The debug OS21 libraries (selected with `-mruntime=os21_d`) are built with `CONF_DEBUG` defined. The OS21 libraries selected when the `-m4-nofpu` compiler option is specified have also been built with `CONF_NO_FPU_SUPPORT` defined.

CONF_DEBUG

To produce a debug OS21 kernel, define the `CONF_DEBUG` preprocessor symbol. This kernel contains many checks to ensure internal integrity, and to check that user calls into the kernel are correct.

CONF_DEBUG_ALLOC

To produce an OS21 kernel with special checks added to the memory management code (including the detection of heap scribbles, and the freeing of bad pointers), define the `CONF_DEBUG_ALLOC` preprocessor symbol.

CONF_DEBUG_CHECK_SEM, CONF_DEBUG_CHECK_MTX and CONF_DEBUG_CHECK_EVT

To produce an OS21 kernel with extra integrity checks enabled for semaphores, mutexes and event flags respectively, define these preprocessor symbols. Every time one of these objects is referenced, OS21 performs extra checks to ensure that its structure is not corrupt, and that it has not been previously deleted.

CONF_FINE_GRAIN_CLOCK

Defining this preprocessor symbol causes OS21 to program the system clock to operate at as high a frequency as possible, given the prevailing timer input clock. This increases the number of ticks per second, and hence yields greater accuracy when reading the system time, or setting timeouts.

CONF_FPU_SINGLE_BANK

Defining this preprocessor symbol produces an OS21 kernel with a restricted FPU context save and restore. OS21 only considers the bank of FPU registers which are used by GCC when saving or restoring a context. This halves the number of FPU registers which need to be saved or restored on context switch, therefore improving context switch time and interrupt latency. This option is safe to use provided that no custom FPU routines which perform FPU bank switching are being used.

CONF_INLINE_FUNCTIONS

To produce an OS21 kernel with inlined list manipulation functions, define the `CONF_INLINE_FUNCTIONS` preprocessor symbol. This can yield a slight performance improvement.

CONF_NO_FPU_SUPPORT

Defining this preprocessor symbol produces an OS21 kernel which disregards the FPU registers when switching context. This improves context switch time and interrupt latency, at the expense of making the use of the FPU task unsafe. It is therefore important that the program is built with the `-m4-nofpu` compiler option to ensure that the FPU is not used at all, or that the FPU is just used by a single task.

9.2 Building the OS21 board support libraries

The OS21 board support libraries can be built by invoking `make` from the root of the OS21 source tree (`sh-superh-elf/src/os21`) with the target `buildbsp`. Each board support library consists of four object files:

- one for generic configuration options and for functions that allow the user to insert code to be executed at certain key OS21 events
- one for ST40 core support
- one for SoC support
- one for target board support

The board support source code is located in the subdirectory `sh-superh-elf/src/os21/src/st40/bsp` of the release installation directory.

For more information about the board support libraries, see the chapter entitled “Board support package” in *OS21 User manual* (7358306). There is also information related to the board support libraries in *OS21 for ST40 User manual* (7358673).

Core support files

There is a core support file for each supported ST40 variant. The convention for naming these files is `cpu_`*`variant`*`.c`, where *`variant`* is the name of the SoC. For example, the name of the support file for the ST40 in the STb7100 is `cpu_stb7100.c`.

A full list of all the core support files can be obtained by listing the contents of the `sh-superh-elf/src/os21/src/st40/bsp` directory.

Each file contains:

- declarations of all the interrupts that can be serviced on this core
- tables enumerating all the interrupts and interrupt groups
- base addresses of the interrupt controllers on this core
- interrupt controller initialization flags
- a function, `bsp_cpu_type()`, to return the name of the core

Platform specific variants of the core support files may also exist to support platforms where the external interrupts are not priority encoded (the default).

SoC support files

There is a SoC support file for each supported ST40 system-on-a-chip device. The convention for naming these files is `chip_`*`variant`*`.c`, where *`variant`* is the name of the system-on-a-chip device. For example, the name of the support file for the STb7100 is `chip_stb7100.c`.

A full list of all the SoC support files can be obtained by listing the contents of the `sh-superh-elf/src/os21/src/st40/bsp` directory.

Each file contains:

- a function, `bsp_chip_type()`, to return the name of the SoC
- optionally, the function `bsp_timer_input_clock_frequency_hz()`, and associated support code to determine the clock frequencies of the part from the on-chip clock peripheral

Target board support files

There is a target board support file for each supported reference platform. The convention for naming these files is `board_`*`platform`*`.c`, where *`platform`* is the name of the reference platform. For example, the name of the support file for the STb7100-MBoard is `board_mb411.c`.

A full list of all the board support files can be obtained by listing the contents of the `sh-superh-elf/src/os21/src/st40/bsp` directory.

Each file contains a function, `bsp_board_type()`, that OS21 calls to determine the name of the target board and a variable that declares the frequency of the external crystal on the board.

9.2.1 Creating a customized board support library

The following steps show the process of creating a new board support library for a fictitious board called **custom**.

1. In the `sh-superh-elf/src/os21/src/st40/bsp` directory, copy one of the supplied target board support files to use as a template. For example, copy `board_mb411.c` to `board_custom.c`.
2. Edit it to meet the specific requirements of the new board.
3. In the `sh-superh-elf/src/os21` directory, edit `makest40bsp.inc`.
   a) Declare a new library name:

   ```
   BSPLIB_CUSTOM = $(OS21LIBDIR)/libcustom$(LIBSFX).a
   ```

   b) Add `$(BSPLIB_CUSTOM)` to the list defined by `BSPLIBS`.

   c) Create a definition of the object files to be put in this library. Choose the appropriate CPU support file, based on the processor used on the board, for example:

   ```
   BSPOBJS_CUSTOM = \
     $(OS21OBJDIR)/os21/bsp/bsp$(OBJSFX).o \
     $(OS21OBJDIR)/st40/bsp/cpu_stb7100$(OBJSFX).o \
     $(OS21OBJDIR)/st40/bsp/chip_stb7100$(OBJSFX).o \
     $(OS21OBJDIR)/st40/bsp/board_custom$(OBJSFX).o
   ```

   d) Add `$(BSPOBJS_CUSTOM)` to the list `BSPOBJS`.

   e) Add a rule to build the new library:

   ```
   $(BSPLIB_CUSTOM): $(BSPOBJS_CUSTOM)
     $(ARBUILD) $(BSPOBJS_CUSTOM)
     $(RANLIB) $@
   ```

4. Build the board support library by invoking `make` from the top level OS21 directory, with the target `buildbsp`.

Since by default `LIBSFX` is empty, the resulting library name is `libcustom.a`. It is placed in the `sh-superh-elf/src/os21/lib/st40` directory.

9.2.2 Using the built libraries

After rebuilding the OS21 libraries, the `-L` compiler option must be specified to ensure that the linker picks up the new versions of the libraries:

```
sh4gcc -Linstall-directory/sh-superh-elf/src/os21/lib/st40
```

Note: *Where `install-directory` is the location of the release installation directory.*

Debugging the libraries

To debug the libraries using GDB, add the following to the GDB initialization file (`.shgdbinit`):

```
directory install-directory/sh-superh-elf/src/os21
```

This ensures that GDB finds the OS21 source files. If required, optimization can be switched off or reduced when rebuilding OS21 by specifying `-O0` or `-O1` as the optimization level, for example:

```
make build CCOPTFLAGS=-O0
```

9.3 Adding support for new boards

When a new board support library has been built as described in *Section 9.2.1*, it has to be registered with the toolset. This means creating a new GCC compiler **specs** file to describe the memory layout of the board, and to inform the compiler driver of the new board support library. This specs file can be placed in either:

- the working directory
- the `sh-superh-elf/lib/gccscripts` subdirectory of the release installation directory
- in a directory referenced by the `-B` GCC compiler option

Note: *Placing the file in the release installation directory makes it available from wherever the compiler is invoked.*

Continuing the example in *Section 9.2.1*, create a specs file called `customspecs` in one of the above directories, with the following contents:

```
%rename board_link board_link_custom

*_custom:
--defsym .reservedramsize=0x1000 \
--defsym .physrambase=0x04000000 --defsym ___ramsize=0x02000000 \
--defsym .physrombase=0x00000000 --defsym ___romsize=0x00800000

*customp0:
%(_custom) %(region_p0) %(define_29bit_mem)

*customp1:
%(_custom) %(region_p1) %(define_29bit_mem)

*customp2:
%(_custom) %(region_p2) %(define_29bit_mem)

*customp3:
%(_custom) %(region_p3) %(define_29bit_mem)

*board_link:
```

```
%{\
mboard=customp0|mboard=customsimp0:%(customp0);\
mboard=customp1|mboard=customsimp1|mboard=custom|mboard=customsim:%(customp1);\
mboard=customp2|mboard=customsimp2:%(customp2);\
mboard=customp3|mboard=customsimp3:%(customp3);\
mboard=custom*:%e-mboard=custom* unrecognised;\
\
:%(board_link_custom)\
}

%rename lib_os21bsp_base lib_os21bsp_base_custom

*lib_os21bsp_base:
%{\
mboard=customsim*:simbsp;\
mboard=custom*:custom;\
\
:%(lib_os21bsp_base_custom)\
}
```

Where *simbsp* is the name of the OS21 simulator BSP for the SoC used in the board. For example, use stb7100chess for an STb7100 SoC.

The specs file is designed to complement (and not replace) the boards already supported by the toolset. This is achieved using the %rename specs file directive. This renames the default toolset board_link and lib_os21bsp_base specifications and then refers to them in the updated definitions.

The updated definitions also use the if-else conditional GCC specs rule to ensure that the renamed default specifications are only applied if the -mboard option does not specify the platform being added using the supplementary specs file.

Note: *See Section 3.5.2: Linker board support on page 45 for the definition of the* region_px *and* define_29bit_mem *spec strings in the above example. Compiler specs files are very sensitive to spaces and blank lines; make sure that the specs file looks exactly like the example above.*

To use this new specs file, invoke the compiler with the option -specs=customspecs, along with an appropriate -mboard option, for example -mboard=customp1.

It is now possible to create OS21 applications targeted for the new board. For example:

```
sh4gcc -specs=customspecs -o hello.out hello.c -mruntime=os21 -mboard=customp1
```

Note: *Refer to Section 3.5.1: GCC board support setup on page 42 and to the* boardspecs *file to see how the above specs file may be extended to add support for 32 bit Space Enhancement mode.*

9.4 GDB OS21 awareness support

GDB provides OS21 task aware debugging with the **shtdi** GDB target. The **shtdi** target installs a service which runs on the host and has knowledge of the data structures used in the OS21 kernel. A dependency therefore exists between the version of OS21 being used and the version of the **shtdi** service being used.

OS21 is built with static data tables which expose the layout of certain critical data structures to the **shtdi** service. Each data table has a cyclic redundancy check (CRC) calculated for it, and this too is stored statically. These data tables are auto-generated as part of the OS21 build process. At the same time a header file is also auto-generated which is imported into the build of the **shtdi** service. This header file contains the same CRC values, and some key type definitions.

The data tables are offset/size pairs which identify particular fields within OS21 data structures. The tables are indexed by enumerated types, which are the types imported by the **shtdi** service. There is one data table per OS21 data structure type of which the **shtdi** service has to be aware. The CRC value for each table is calculated using the field name, and since it is a CRC, the order in which the fields appear relative to each other is important. If a field changes name between releases, or fields alter position within a data structure (relative to each other), then the CRC for the data table also changes.

When the **shtdi** service examines a target system to determine if it can debug the target in OS21 aware mode, it examines the data table CRCs in memory and checks to see if they match the ones it was built with. If they do, then OS21 awareness is enabled, and the **shtdi** service can use the in-memory data tables to determine how to parse the OS21 data structures. If the CRCs do not match, then the **shtdi** service and OS21 were not built from the same source base, and the **shtdi** service cannot debug the target in OS21 aware mode.

When modifying OS21, take note that changing the relative order of certain fields in key data structures, or renaming them, may render the **shtdi** service unable to debug the resulting OS21 executables in OS21 aware mode.

9.4.1 Generation of the shtdi service data tables

The following Perl script (invoked automatically as part of the build process) generates the data required by the **shtdi** service:

`sh-superh-elf/src/os21/scripts/mkgdb.pl`

This Perl script is passed key OS21 header files, which are scanned for special mark-ups. These mark-ups identify which structures, and which fields in those structures, are to be exposed to the **shtdi** service. The mark-ups used are very simple, and are designed to be invisible to the C compiler when the headers are compiled, by using the C preprocessor.

`_OS21_GDB_STRUCT(`*`struct`*`)`

Declares this structure as containing information required by the **shtdi** service. This decoration triggers the generation of the following data objects:

- a `size_t`, with the size of the structure, given the name *`struct`*`_size`
- an array of offset and size descriptors, given the name *`struct`*`_descs`
- a `size_t`, with the number of elements in the above array, given the name *`struct`*`_descs_size`
- an `unsigned int` with the value of the calculated CRC for the above array, given the name *`struct`*`_descs_crc`

`_OS21_GDB_FIELD(`*`level`*`, `*`enum_prefix`*`, `*`field`*`)`

Declares that a field in the current structure is to be exposed to the **shtdi** service, where *`level`* is the **shtdi** compatability level, starting from 0 for the default. An enum called *`enum_prefix_field`* is generated and stored in the export header file and corresponds to this field’s index in the array of descriptors.

`_OS21_GDB_ARRAY_FIELD(`*`level`*`, `*`enum_prefix`*`, `*`field`*`, `*`field_index`*`,`
`    `*`enum_suffix`*`)`

Declares that a particular field in the structure array is to be exposed to the **shtdi** service, where *`level`* is the **shtdi** compatability level, starting from 0 for the default. An enum called *`enum_prefix_field_enum_suffix`* is generated and stored in the export header file and corresponds to this field’s index in the array of descriptors.

`_OS21_GDB_BEGIN_EXPORT, _OS21_GDB_END_EXPORT`

These two markers are used to identify a section of header file which is to be copied verbatim into the export header file.

10 Booting OS21 from Flash ROM

Examples of booting from Flash ROM are located in the `sh-superh-elf/examples/os21` subdirectory of the release installation directory. Full details can be found in the `readme.txt` files in the example directories. *Table 33* lists the Flash ROM examples.

Table 33. Examples of booting from Flash ROM

Example name	Description	SE Mode
`failsafe`	Creates a fail-safe (that is, integrity checking and, potentially, repairing) application in Flash ROM.	?
`nandboot`	Example of booting from NAND Flash ROM. It also demonstrates how to create a Flash file system, using OSPlus, and load and execute applications from that file system.	?
`nandbootblock0`	A second example of booting from NAND Flash ROM and creating a Flash file system using OSPlus. This is for older ST NAND controllers that do not support multiple block remapping in boot mode. See *Section 10.3 on page 138*.	?
`rombootanywhere`	Demonstrates how to link an application that has some of its code and data placed in an alternative memory location.	?[???]
`rombootram`	The Flash ROM bootstrap copies the application to RAM before running.	?
`rombootrom`	The application runs directly from Flash ROM.	?
`romdynamic`	The application boots from Flash ROM and uses the Relocatable Loader Library to load and call a dynamic library from Flash ROM.	?
`rommultiboot`	The ST40 and ST231 cores on an SoC (other than a STi5528) boot from Flash ROM.	?
`sti5528dualboot`	The ST40 and ST20 cores on the STi5528 boot from Flash ROM.	?

1. If the application is located in Flash ROM, this example does not support SE mode.

The examples `romloader` and `sti5528loader` also show how to generate a ROM image for booting the co-processors only under the control of a separately downloaded ST40 application.

10.1 Overview of booting from Flash ROM

The ST40 Micro Toolset supports both single core and multicore SoCs where each core boots from the same Flash ROM. This means that the contents of Flash ROM must be managed so that multiple code and data images can co-exist and, where necessary, can be separately updated.

10.2 Standard Flash ROM layout

The OS21 examples in the ST40 Micro Toolset that boot from Flash ROM include tools for laying out the Flash ROM. For most SoCs, the tools lay out the Flash ROM in the same way. One exception is the STi5528, which has a 1024 byte reserved region at the start of the Flash ROM, followed by the usual layout. There is also a specialized layout for NAND Flash ROM for those SoCs that do not have an ST NAND controller supporting multiple block remapping in boot mode (see *Section 10.3 on page 138* for more information).

The layout used by the tools in the examples supports:

- boot vectors for up to eight cores at 0x40 byte offsets from the base of the Flash ROM
- bootstrap information for up to eight cores
- an optional fail-safe application which can run before the main applications to check the integrity of the Flash ROM and report/fix any problems
- a main application image directory
- main application image-control structures for images in the directory, which point to the code and data sections located throughout the rest of the Flash ROM

More detail on the Flash ROM layout used can be found in the comments near the start of the `flasher.c` or `mkbinrom.pl` source files found in the Flash ROM example directories (see *Table 33 on page 135*).

Applications can be placed in NOR Flash ROM by the **flasher** Flash ROM programming tool included in the examples. The **flasher** tool can either take component image files (for example, for bootvectors, bootstraps or applications), or a complete Flash ROM image file such as that created by the `mkbinrom.pl` tool, as its input.

The **flasher** tool may also be combined with a Flash ROM image to create a self-flashing executable tool that programs the NOR Flash ROM from the embedded image.

Note: All NOR Flash ROM examples create a self-flashing executable tool.

The **nandflasher** tool is provided in the examples that demonstrate booting from NAND Flash ROM. The **nandflasher** tool requires a binary ROM image created using the `mkbinrom.pl` tool when using the standard Flash ROM layout.

Most examples that boot from Flash ROM also include a tool called **flashdir** which can display the contents of Flash ROM.

The `mkimage.pl` tool converts target executable files into the component image file format. It can process ST40 and ST200 ELF files (executable files and relocatable libraries), as well as ST20 hex and S-record ROM format files. Executable files and relocatable libraries are converted into their constituent sections, which are placed in the component image file.

Note: If `mkimage.pl` *is used to create an image file from an ST40 ELF file that has previously been stripped of its symbol information,* `mkimage.pl` *fails to find the address of the stack and several other important symbols. In order to avoid this problem, use the option* `--`

`keep-symbol` *when stripping symbol information, and specify the symbols that* `mkimage.pl` *requires to operate correctly. For an ST40 ELF file, these symbols are:*

- `_SH_DEBUGGER_CONNECTED`
- `_stack`
- `.physrambase`
- `.rombaseoffset`
- `___rambase`
- `___ramsize`

Using the **flasher** tool with component image files allows updates to the existing contents of Flash ROM. Any component image file being placed in Flash ROM results in an update to the image directory pages. Multiple application images can be stored in Flash ROM. Each image is tagged in the image directory with its associated core. A core can have multiple application images stored in Flash ROM, but only one is tagged as the boot application for that core.

The `mkbinrom.pl` tool provided with the examples creates a complete binary Flash ROM image in a single file. Like the **flasher** tool it can take component image files as input, but it can also take executable and relocatable library ELF files directly (in which case it calls `mkimage.pl` automatically to convert them to component image files). The **flasher** tool or the **nandflasher** tool can then be used to program the binary image to Flash ROM. Complete Flash ROM images cannot be updated, but must be re-created from all constituent ELF files and/or component image files each time.

The examples provide sample boot vector and bootstrap code for the ST40, which is able to locate the ST40 application in the Flash ROM, and start it. The flow of execution on booting is as follows:

1. the ST40 boot vector code runs, jumping to the ST40 bootstrap code
2. the ST40 bootstrap code then:
   - configures the clocks, EMI and LMI interfaces
   - configures the platform for 32-bit Space Enhancement mode (if required)
   - locates the ST40 boot application (or a fail-safe application if one is present)
   - moves any sections to RAM that require moving
   - zeros any sections in RAM that require zeroing
   - transfers control to the ST40 fail-safe or main application

If a fail-safe application is present, it is responsible for loading the main ST40 boot application. A simple example fail-safe application is included in the `failsafe` example.

10.2.1 romgen creation of bootstrap code

A tool called **romgen** can be used to assist in the creation of the bootstrap code for configuring an SoC. The **romgen** tool converts an ST TargetPack into a set of ROM initialization operations that can be included in the assembly language bootstrap code. The mechanism for doing this is described in the `readme.txt` file supplied with the `romboottram` example.

The **romgen** tool is part of the ST Micro Connection Package and is described in detail in the *ST TargetPack User manual* (8020851), provided with the ST Micro Connection Package.

10.3 NAND Flash ROM layout

The more recent ST NAND controllers allow a standard Flash ROM layout to be used (see *Section 10.2 on page 136*). A special NAND Flash ROM layout is provided for SoCs that have an older ST NAND controller that does not support multiple block remapping in boot mode. In this case, only block 0 of the NAND Flash can be relied upon to be reliable without further software checking. This means that the first boot stage must be from a Flash ROM boot image that fits within a single NAND Flash block (this is typically 16KB for small page NAND Flash, or 128KB for large page NAND Flash).

The layout is closely related to the standard Flash ROM layout in some respects, but has been modified and extended to cope with the special requirements of NAND Flash ROM. The layout is designed to take advantage of the NAND Flash ROM file-system support available in OSPlus.[a] (For information about OSPlus, see *Section 2.2: OSPlus on page 33*.)

For SoCs that have an ST NAND controller that supports multiple block remapping in boot mode, STMicroelectronics recommends using the standard Flash ROM layout, as demonstrated in the `nandboot` example.

The layout used by the tools in the `nandbootblock0` example supports:

- a boot vector for a single core (space is reserved for up to four, but this is not required for any current NAND Flash ROM enabled SoC)
- a two-stage bootstrap (implemented in assembler and C)
- a payload application
- an OSPlus file-system containing multiple application images for an arbitrary number of cores

More details on the NAND Flash ROM layout can be found in the `readme.txt` file of the `nandbootblock0` example.

Applications can be placed in NAND Flash ROM by the **nandflasher** NAND Flash ROM programming tool included in the `nandbootblock0` example. When using the NAND Flash layout, the **nandflasher** tool does not write to fixed locations in NAND Flash ROM, but rather selects blocks in the NAND Flash ROM that are known to work. The bootstrap code contains an algorithm to find the data at run time. Because of the dynamic nature of this Flash ROM layout, there is no way to create a complete binary Flash ROM image. Use the `mknandboot.pl` tool for creating a binary image of the contents of block 0 of the NAND Flash ROM.

Note: Note: Unlike the NOR Flash ROM boot examples, no self-flashing executable tool is available.

The NAND Flash ROM layout uses component image files of the same format used by the NOR Flash ROM examples. These are generated using the `mkimage.pl` tool. See *Section 10.2 on page 136*.

[a] Booting from NAND Flash ROM does not introduce a dependency on OSPlus for the main application.

11 Relocatable loader library

The relocatable loader library (**rl_lib**) provides support for the creation and loading of dynamic shared objects (DSOs) in an embedded environment. **rl_lib** implements DSOs (also known as load modules or relocatable libraries) as defined in the standard for supporting ELF System V Dynamic Linking.

The Unix System V generic ABI may be found at www.sco.com/developers/devspecs.

This relocatable loader library only supports OS21 applications. There is no relocatable loader support for bare machine applications.

11.1 Run-time model overview

The ELF System V ABI supports several run-time models, of which only some are suitable for embedded systems without the support of traditional operating system services. The run-time model for an application dictates the method used for linking and loading. *Table 34* lists the different run-time models and *Table 35* summarizes the features supported by each model.

rl_lib only implements the **R_Relocatable** run-time model.

Table 34. Run-time models

Run-time model	Description
R_Absolute	Absolute run-time model. The application is a single module that is statically linked at a fixed load address.
R_Relocatable	Relocatable run-time model. The application has a main module and several load modules. The main module is statically linked and loaded as for an **R_Absolute** application. The load modules are loaded on demand (by explicit calls to the loader) at run-time. The load modules are loaded at an arbitrary address and dynamic symbol binding is applied by the loader for symbols undefined in the load modules. The dynamic symbol binding traverses the modules bottom up in the hierarchy of loaded modules. See *Section 11.2* for details.
R_PIC	System V run-time model. The application has a main module and several load modules. The main module is typically statically linked but may have references to symbols in the load modules. The main module is loaded with support from the dynamic loader that also loads load modules and binds symbols before the application starts. At run-time, the application may also load other modules on demand. The dynamic symbol binding walks the load modules in an order which is defined by the static link order and the run-time loading order. In addition to dynamic loading and linking, the load module’s segments can be shared between several applications in a multi-process environment. This model usually relies on file system support and virtual memory management.

Table 35. Run-time model features

	R_Absolute	R_Relocatable	R_PIC
Application partitioning	1 single program	1 main module + N load modules	1 main module + N load modules
Static symbol binding	Yes	Yes	Yes
Dynamic loading	No	Startup time: No Run-time: Yes	Startup time: Yes Run-time: Yes
Dynamic symbol binding	No	Main module: No Load modules: Yes	Main module: Yes Load modules: Yes
Explicit module dependencies	N/A	No	Yes
Dynamic symbol lookup	N/A	Bottom up (from loaded to loader)	Unrestricted order
Symbol preemption	N/A	No	Yes
Segment sharing (across processes)	N/A	No	Yes
Performance impact	N/A	Minimal	Yes
Code size impact	N/A	Minimal	Yes
Application writer impact	N/A	Need explicit loading	No change by default
Build system impact	N/A	Compiler options. Load modules build.	Compiler options. Load modules build.

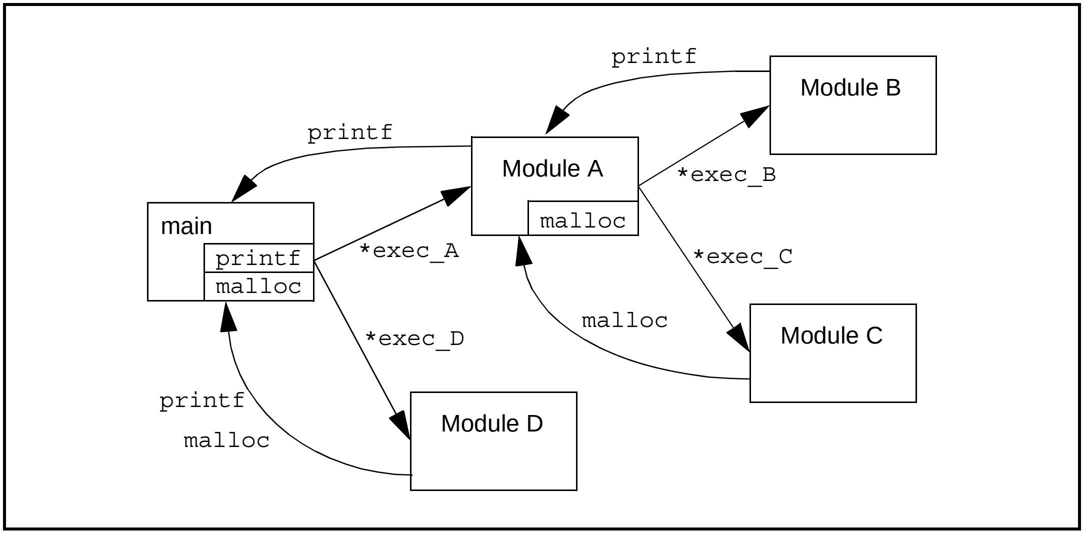
11.2 Relocatable run-time model

The **R_Relocatable** run-time model as implemented by **rl_lib** has the following features:

- one main module loaded at application startup by the system
- several load modules that can be loaded at run-time and unloaded after use
- several modules can be resident at the same time
- a loaded module, as for the main module, can load and unload other load modules
- load modules can be loaded anywhere
- access to symbols in loaded modules from the loader through a call to the loader library
- dynamic symbol binding is performed by the loader when loading a module and symbols are searched in the load modules hierarchy bottom-up (to the main module)
- sharing of code and data objects between modules is achieved by linking to the objects in a common ancestor
- the loader library is statically linked with the main module
- generally, system support archive library should be linked with the main module

Figure 26 shows an example of application that has four load modules A, B, C and D.

Figure 26. Example of an application with four load modules



In *Figure 26*, curved arrows (from load modules to parent module) represent load time symbol binding performed while the load module is loaded and straight arrows (from loader module to loaded module) represent explicit symbol address resolution performed through the loader library API.

The following describes a possible scenario.

1. At run-time the main module loads into memory the module A using the `rl_load_file()` function.
2. The loader, in the process of loading A into memory, binds the symbol `printf` (undefined in A) to the `printf` function defined in main.
3. The main module uses the `rl_sym()` function to retrieve a pointer to the function symbol `exec_A` in A.
4. For A, the main program loads the module D and references to `printf` are resolved to the `printf` in main. In addition, references to `malloc` in D are also resolved to the `malloc` in main.
5. The main module retrieves a pointer to `exec_D` in D using the `rl_sym()` function.
6. The main module (at some point) invokes the function `exec_A`.
7. The `exec_A` function loads a further two modules B and C.
8. The undefined reference to `printf` in B is resolved to the `printf` in main (the loader searches first in A, and then in main).
9. The undefined reference to `malloc` in C is resolved to the `malloc` in A (the loader searches for and finds it in A). Note that the `malloc` function called from D (`malloc` of main) is different from the `malloc` function called from B (or C, or A) which is the `malloc` of A.
10. After retrieving symbol addresses using the `rl_sym()` function, module A can indirectly call functions or reference data in B and C .

Note: *At any time, the main module or the module A may unload one of the loaded modules.*

11.2.1 The relocatable code generation model

The relocatable code generation model is the same as the code generation model for the System V model with the following differences.

- No symbol can be preempted. Dynamic symbol binding always searches the current module first. This has the effect that a module containing a symbol definition can be sure that it will use this definition. This allows inlining in load modules for example.
- Weak references are treated the same way as undefined references in load modules. Therefore, when traversing the module tree bottom-up, the first definition found is taken.

11.3 Relocatable loader library API

The relocatable loader library provides support for loading and unloading a module and for accessing a symbol address in a module by name. The relocatable loader library is provided as a library `librl.a` and its associated header file `rl_lib.h`.

The functions defined in this API are explained in the following sections.

rl_handle_t type

All the functions manipulating a load module use a pointer to the `rl_handle_t` type. This is an abstract type for a load module handle.

A load module handle is allocated by the `rl_handle_new()` function and deallocated by the `rl_handle_delete()` function.

The main module handle is statically allocated and initialized in the startup code of the main module.

A module handle references one loaded module at a time. To load another module from the same handle, the previous module must first be unloaded.

rl_handle_new

Allocate and initialize a new handle

Definition:

```
rl_handle_t *rl_handle_new(
  const rl_handle_t *parent,
  int mode);
```

Arguments:

`parent`	The handle of the parent module.
`mode`	Reserved for future extensions.

Returns: The newly initialized handle or `NULL` on failure.

Description: The `rl_handle_new()` function allocates and initializes a new handle that can be used for loading and unloading a load module.

The handle of the parent module to which the loaded module is connected is specified by the `parent` argument.

The `mode` argument is reserved for future extensions and must be `0`.

Generally, a load module will be attached to the module using this function, therefore a handle will typically be allocated as follows:

```
rl_handle_t *new_handle = rl_handle_new(rl_this(), 0);
```

rl_handle_delete

Finalize and deallocate a module handle

Definition:

```
int rl_handle_delete(
  rl_handle_t *handle);
```

Arguments:

`handle`	The handle to deallocate.

Returns: Returns `0` for success, `-1` for failure.

Description: The `rl_handle_delete()` function finalizes and deallocates a module handle.

The handle must not hold a loaded module. The loaded module must have been first unloaded by `rl_unload()` before calling this function. If successful, the value returned is `0`. Otherwise, the value returned is `-1` and the error code returned by `rl_errno()` is set accordingly.

rl_this

Return the handle for the current module

Definition:

```
rl_handle_t *rl_this(void);
```

Arguments: None.

Returns: The handle for the current module.

Description: The `rl_this()` function returns the handle for the current module. If called from the main module, it returns the handle of the main module. If called from a loaded module, it returns the handle that holds the loaded module.

This function is used when allocating a handle with `rl_handle_new()`. It can also be used, for example, to retrieve a symbol in the current module:

```
void *symbolPtr = rl_sym(rl_this(), "_symbol");
```

rl_parent Return the handle for the parent of the current handle

Definition:

```
rl_handle_t *rl_parent(void);
```

Arguments: None.

Returns: The handle for the parent of the current handle, or `NULL` if there is no parent module.

Description: The `rl_parent()` function returns the handle for the parent of the current handle (as returned by `rl_this()`).

It may be used, for example, to find a symbol in one of the parent modules:

```
void *parentSymbolPtr = rl_sym_rec(rl_parent(), "_symbol");
```

rl_load_addr Return the memory load address of a loaded module

Definition:

```
const char *rl_load_addr(
  rl_handle_t *handle);
```

Arguments:

`handle`	The handle for the loaded module.

Returns: The memory load address of the loaded module, or `NULL`.

Description: The `rl_load_addr()` function returns the memory load address of a loaded module. It returns `NULL` if the handle does not hold a loaded module or if the handle passed is the main module handle.

rl_load_size Return the memory load size of a loaded module

Definition:

```
unsigned int rl_load_size(
  rl_handle_t *handle);
```

Arguments:

`handle`	The handle for the loaded module.

Returns: The memory load size of the loaded module, or `0`.

Description: The `rl_load_size()` function returns the memory load size of a loaded module. It returns `0` if the handle does not hold a loaded module or if the handle passed is the main module handle.

rl_file_name Return the file name associated with the loaded module handle

Definition:

```
const char *rl_file_name(
  rl_handle_t *handle);
```

Arguments:

`handle`	The handle for the loaded module.

Returns: The file name associated with the loaded module handle, or `NULL`.

Description: The `rl_file_name()` function returns the file name associated with the loaded module handle. It returns `NULL` if no file name is associated with the current loaded module, if the handle does not hold a loaded module or if the handle passed is the main program handle.

rl_set_file_name — Specify a file name for the handle

Definition:

```
int rl_set_file_name(
  rl_handle_t *handle,
  const char *f_name);
```

Arguments:

`handle`	The handle for the module.
`f_name`	The file name to specify for the handle.

Returns: Returns `0` for success, `-1` for failure.

Description: The `rl_set_file_name()` function is used to specify a file name for the `handle`. The file name is attached to the next module that will be loaded. It can be used to specify a file name for modules loaded from memory or a byte stream, or to force a different file name for a module loaded from a file.

This function returns `0` if the file name was successfully set, or `-1` and the error code returned by `rl_errno()` is set accordingly if the module is already loaded or if the application runs out of memory.

rl_load_buffer — Load a load module into memory

Definition:

```
int rl_load_buffer(
  rl_handle_t *handle,
  const char *image);
```

Arguments:

`handle`	The handle for the module.
`image`	The image of the load module.

Returns: Returns `0` for success, `-1` for failure.

Description: The `rl_load_buffer()` function loads a load module into memory from the image referenced by `image`.

The function allocates the space for the loaded module from the heap, loads the segments from the memory image of the loadable module, links the module to the parent module of the handle and relocates and initializes the loaded module.

This function calls the action callback functions for the `RL_ACTION_LOAD` action after loading and before executing any code in the loaded module.

`0` is returned if the loading was successful, `-1` is returned on failure and the error code returned by `rl_errno()` is set accordingly.

rl_load_file   Load a load module into memory from a file

Definition:

```
int rl_load_file(
  rl_handle_t *handle,
  const char *f_name);
```

Arguments:

`handle`	The handle for the module.
`f_name`	The file from which to load the load module.

Returns: Returns `0` for success, `-1` for failure.

Description: The `rl_load_file()` function loads a load module into memory from the file specified by `f_name`.

This function opens the specified file with an `fopen()` call, allocates space for the loaded module from the heap, loads the segments from the file, links the module to the parent module of the handle, relocates and initializes the loaded module. The file is closed with `fclose()` before returning. This function calls the action callback functions for the `RL_ACTION_LOAD` action after loading and before executing any code in the loaded module.

`0` is returned if the loading was successful, `-1` is returned on failure and the error code returned by `rl_errno()` is set accordingly.

rl_load_stream Load a load module into memory from a byte stream

Definition:

```
typedef int rl_stream_func_t(
  void *cookie,
  char *buffer,
  int length);

int rl_load_stream(
  rl_handle_t *handle,
  rl_stream_func_t *stream_func,
  void *stream_cookie);
```

Arguments:

`handle`	The handle for the module.
`stream_func`	The user specified callback function.
`stream_cookie`	The user specified state.

Returns: Returns `0` for success, `-1` for failure.

Description: The `rl_load_stream()` function loads a load module into memory from a byte stream provided by a user specified callback function `stream_func` and the user specified state `stream_cookie`.

The callback function must be of type `rl_stream_func_t`. It is called multiple times by the loader to retrieve the load module data into the buffer `buffer` of length `length` until the module is loaded into memory. The loader always calls the callback function with a buffer length strictly greater than 0. The `stream_cookie` argument passed to `rl_load_stream()` is passed to the callback function in its `cookie` parameter. The `cookie` parameter is intended to be used by the callback function to access private state.

The callback function must return the number of bytes transferred. If the returned value is less than the specified buffer length or is `-1`, `rl_load_stream()` will in turn return an error and the error code returned by `rl_errno()` is set accordingly.

The `rl_load_stream()` function allocates the space for the loaded module from the heap, loads the segments by calling the callback function, links the module to the parent module of the handle, relocates and initializes the loaded module. This function calls the action callback functions for the `RL_ACTION_LOAD` action after loading and before executing any code in the loaded module.

`0` is returned if the loading was successful, `-1` is returned on failure and the error code returned by `rl_errno()` is set accordingly.

This function can be used as an alternative to `rl_load_buffer()` or `rl_load_file()` to allow any loading method to be implemented.

The following example illustrates how the `rl_load_file()` function may be implemented using the `rl_load_stream()` function:

```
/* User implementation of the callback function that reads from
a file. */
static int rl_stream_read(FILE *file, char *buffer, int length)
{
   int nbytes;
   nbytes = fread(buffer, 1, length, file);
```

```
  return nbytes;
}
...
{
  /* Loads the module from a file.*/
  FILE *file;
  int status;
  file = fopen(f_name, "rb");
  if (file == NULL) { /* ... error ...*/ }
  status = rl_load_stream(handle,
           (rl_stream_func_t *)rl_stream_read, file);
  if (status == -1) { /* ... error ...*/ }
  fclose(file);
}
...
```

rl_unload

Unload a previously loaded load module

Definition:

```
int rl_unload(
  rl_handle_t *handle);
```

Arguments:

handle	The handle for the module.

Returns: Returns 0 for success, -1 for failure.

Description: The rl_unload() function unloads a previously loaded load module. It finalizes, unlinks, and frees allocated memory for the loaded module. This function calls the action callback functions for the RL_ACTION_UNLOAD action before unloading and after having executed finalization code in the module.

The return value is 0 if the unloading is successful, otherwise the return value is -1 and the error code returned by rl_errno() is set accordingly.

rl_sym Return a pointer reference to a symbol in the loaded module

Definition:

```
void *rl_sym(
  rl_handle_t *handle,
  const char *name);
```

Arguments:

`handle`	The handle for the loaded module.
`name`	The symbol in the loaded module.

Returns: The pointer reference to the symbol.

Description: The `rl_sym()` function returns a pointer reference to the symbol `name` in the loaded module specified by `handle`. It searches the dynamic symbol table of the loaded module and returns a pointer to the symbol. The `handle` parameter can be the handle of any currently loaded module, or the handle of the main module.

If the symbol is not defined in the loaded module, `NULL` is returned. It is not generally an error for this function to fail. For example, the user may conditionally call a specific function only if it is defined in the module.

In this function, as well as for the `rl_sym_rec()` function, the `name` parameter must be the mangled symbol name. For ST40 modules, C names are always mangled by prefixing the name with an underscore (_). For example, to return a reference to the `printf()` function, the symbol name passed to `rl_sym()` will be “`_printf`”. Use the **sh4nm** tool to obtain the mangled names of C and C++ symbols. Use **sh4c++filt** to obtain the demangled names.

rl_sym_rec Return a pointer reference to a symbol in the loaded module or one of its ancestors

Definition:

```
void *rl_sym_rec(
  rl_handle_t *handle,
  const char *name);
```

Arguments:

`handle`	The handle for the loaded module.
`name`	The symbol in the loaded module.

Returns: The pointer reference to the symbol.

Description: The `rl_sym_rec()` function returns a pointer reference to the symbol named `name` found in the loaded module specified by `handle` or one of its ancestors.

This function searches the dynamic symbol table of the loaded module and returns a pointer to the symbol if found. If the symbol is not found, the function iteratively searches in the dynamic symbol table of the parent modules until the symbol is found. The `handle` parameter can be the handle of any currently loaded module, or the handle of the main module.

If the symbol is not defined in the loaded module or one of its ancestors, `NULL` is returned.

The `name` parameter must be the mangled symbol name as for the `rl_sym()` function (see *rl_sym on page 149*).

rl_foreach_segment

Iterate over all the segments of loaded module and call the supplied function

Definition:

```
typedef rl_segment_info_t_ rl_segment_info_t;

typedef int rl_segment_func_t (
  rl_handle_t *handle,
  rl_segment_info_t *seg_info,
  void *cookie);

int rl_foreach_segment(
  rl_handle_t *handle,
  rl_segment_func_t *callback_fn,
  void *callback_cookie);
```

Arguments:

`handle`	The handle for the module.
`callback_fn`	The user specified callback function.
`callback_cookie`	The user specified state.

Returns: Returns `0` for success, `-1` for failure.

Description: The `rl_foreach_segment()` function iterates over all the segments of the loaded module specified by `handle` and calls back the user supplied function. For each segment the function `callback_fn` is called with the following parameters:

`handle`	The handle passed to `rl_foreach_segment()`.
`seg_info`	The current segment information.
`cookie`	The `callback_cookie` argument passed to `rl_foreach_segment()`.

The segment information returned in `seg_info` is a pointer to the following structure:

```
typedef unsigned int rl_segment_flag_t;

#define RL_SEG_EXEC 1
#define RL_SEG_WRITE 2
#define RL_SEG_READ 4

struct rl_segment_info_t_ {
  const char *seg_addr;
  unsigned int seg_size;
  rl_segment_flag_t seg_flags;
};
```

The user callback function must return `0` on success or `-1` on error.

In the case where the callback function returns an error, the `rl_foreach_segment()` function returns `-1` and the error code returned by `rl_errno()` is set to `RL_ERR_SEGMENTF`.

rl_add_action_callback — Add a user action callback function to the user action callback list

Definition:

```
typedef unsigned int rl_action_t;

#define RL_ACTION_LOAD 1
#define RL_ACTION_UNLOAD 2
#define RL_ACTION_ALL ((rl_action_t) - 1)

typedef int rl_action_func_t (
  rl_handle_t *handle,
  rl_action_t action,
  void *cookie);

int rl_add_action_callback(
  rl_action_t action_mask,
  rl_action_func_t *callback_fn,
  void *callback_cookie);
```

Arguments:

`action_mask`	The set of actions for which the callback function must be called.
`callback_fn`	The user specified callback function.
`callback_cookie`	The user specified state.

Returns: Returns `0` for success, `-1` for failure.

Description: The `rl_add_action_callback()` function adds a user action callback function to the user action callback list. It can be called any number of times with different callback functions. The same callback function cannot be added more than once.

For each defined action, each callback function is called in the order it was added to the callback list. The callback functions are not attached to a particular module and are called for any loaded/unloaded modules.

This function returns `0` on success and `-1` on failure. No error code is set. This function can fail if a callback function is already in the callback list or if the program runs out of memory.

The type `rl_action_t` defines the action flags for module loading/unloading and is passed to the action function callback. The action flags can be OR-ed to create an action mask which can be passed to the function `rl_add_action_callback()`. The actions defined are:

`RL_ACTION_LOAD`	The callback is called just after the module has been loaded in memory and the cache has been synchronized. No module code has been executed.
`RL_ACTION_UNLOAD`	The callback is called just before the module is unloaded from memory. No module code will be executed after this point.
`RL_ACTION_ALL`	The callback will be called for any of the above actions.

The type for the user action callback function is rl_action_func_t. The parameters passed to the callback function when it is called are:

handle	The handle that performed the action.
action	The action performed.
cookie	The callback_cookie parameter passed to rl_add_action_callback().

The callback function returns 0 on success and -1 on failure. In the case of failure, the loading (or unloading) of the module is undone and the error code returned by rl_errno() is set to RL_ERR_ACTIONF.

rl_delete_action_callback

Remove a user function from the action callback list

Definition:

```
int rl_delete_action_callback(
  rl_action_func_t *callback_fn);
```

Arguments:

callback_fn	The user specified callback function.

Returns: Returns 0 for success, -1 if the callback was not present in the callback list.

Description: The rl_delete_action_callback() function removes the specified callback function from the action callback list. It returns 0 if the callback was removed, or -1 if it was not present in the callback list. No error code is set.

rl_revision

Return the version string of the library

Definition:

```
extern const char * rl_revision(void);
```

Arguments: None.

Returns: A string containing the version of the library.

Description: Use this function to return the version of the library in use.

rl_errno Return the error code for the last failed function

Definition:

```
int rl_errno(
  rl_handle_t *handle);
```

Arguments:

`handle`	The handle for the module.

Returns: The error code for the last failed function.

Description: The `rl_errno()` function returns the error code for the last failed function. *Table 36* lists the possible codes.

Table 36. Errors returned by rl_errno()

Error code	Diagnostic	Possible error causing functions
`RL_ERR_NONE`	No error.	
`RL_ERR_MEM`	Ran out of memory.	`rl_load_buffer()`, `rl_load_file()`, `rl_load_stream()`, `rl_set_file_name()`
`RL_ERR_ELF`	The load module is not a valid ELF file.	`rl_load_buffer()`, `rl_load_file()`, `rl_load_stream()`, `rl_set_file_name()`
`RL_ERR_DYN`	The load module is not a dynamic library.	`rl_load_buffer()`, `rl_load_file()`, `rl_load_stream()`, `rl_set_file_name()`
`RL_ERR_SEG`	The load module has invalid segment information.	`rl_load_buffer()`, `rl_load_file()`, `rl_load_stream()`, `rl_set_file_name()`
`RL_ERR_REL`	The load module contains invalid relocations.	`rl_load_buffer()`, `rl_load_file()`, `rl_load_stream()`, `rl_set_file_name()`
`RL_ERR_RELSYM`	A symbol was not found a load time. `rl_errarg()` returns the symbol name.	`rl_load_buffer()`, `rl_load_file()`, `rl_load_stream()`, `rl_set_file_name()`
`RL_ERR_SYM`	The symbol is not defined in the module. `rl_errarg()` returns the symbol name.	`rl_sym()`, `rl_sym_rec()`
`RL_ERR_FOPEN`	The file cannot be opened by `rl_fopen()`.	`rl_load_file()`
`RL_ERR_FREAD`	Error while reading the file in `rl_fread()`.	`rl_load_file()`
`RL_ERR_STREAM`	Error while loading the file from a stream.	`rl_load_stream()`

Table 36. Errors returned by rl_errno() (continued)

Error code	Diagnostic	Possible error causing functions
RL_ERR_LINKED	Module handle is already linked.	rl_load_file(), rl_load_buffer(), rl_load_stream(), rl_handle_delete()
RL_ERR_NLINKED	Module handle is not yet linked.	rl_unload(),rl_sym(), rl_sym_rec(), rl_foreach_segment()
RL_ERR_SEGMENTF	Error in segment function callback.	rl_foreach_segment()
RL_ERR_ACTIONF	Error in action function callback.	rl_load_file(), rl_load_buffer(), rl_load_stream()

rl_errarg Return the name of the symbol that could not be resolved

Definition:

```
const char *rl_errarg(
  rl_handle_t *handle);
```

Arguments:

handle	The handle for the module.

Returns: The name of the symbol that could not be resolved.

Description: If rl_errno() returns RL_ERR_RELSYM or RL_ERR_SYM, the rl_errarg() function returns the name of the symbol that could not be resolved.

rl_errstr Return a string for an error code

Definition:

```
const char *rl_errstr(
  rl_handle_t *handle);
```

Arguments:

handle	The handle for the module.

Returns: A string for the error code.

Description: The rl_errstr() function returns a string for the error code reported by rl_errno(). For example:

```
...
void *sym = rl_sym(handle, "symbol");
if (sym == NULL) fprintf(stderr, "failed: %s\n", rl_errstr(handle));
...
```

If symbol is not defined in the module referenced by handle then the following message is displayed:

```
failed: symbol not found: symbol
```

11.4 Customization

The relocatable loader library defines a number of functions that it uses internally for providing services such as heap memory management and file access. To provide custom implementations of these functions, the application in the main module may override these functions.

11.4.1 Memory allocation

Table 37 lists the functions used by the relocatable loader library to provide support for heap memory management.

Table 37. Memory allocation functions

API	Default implementation
`void *rl_malloc(int size)`	`malloc(size)`
`void rl_free(void *ptr)`	`free(ptr)`
`void *rl_memalign(int align, int size)`	`memalign(align, size)`
`void rl_memfree(void *ptr, int size)`	`free(ptr)`

Note: *If providing a custom implementation, override all four functions and link the new functions with the main module.*

11.4.2 File management

Table 38 lists the functions used by the `rl_load_file()` function to open, read and close a file handle.

Table 38. File handle functions

API	Default implementation
`void *rl_fopen(const char *f_name, const char *mode)`	`fopen(f_name, mode)`
`int rl_fread(char *buffer, int eltsize, int nelts, void *file)`	`fread(buffer, eltsize, nelts, file)`
`int rl_fclose(void *file)`	`fclose(file)`

Note: *If providing a custom implementation, override all three functions and link them with the main module.*

11.5 Writing and building a relocatable library and main module

11.5.1 Example source code

The following sections use the simple relocatable library, `rl_hello.c`:

```
#include <stdio.h>

void library_function(void)
{
  printf("Hello World!\n");
}
```

The following main module, `main.c`, is used to load the library.

```
#include <stdio.h>
#include <stdlib.h>
#include <os21.h>
#include <rl_lib.h>

int main(void)
{
  rl_handle_t *lib;
  void (*lib_function)();

  kernel_initialize(NULL);
  kernel_start();

  printf("In main program\n");

  lib = rl_handle_new(rl_this(), 0);
  rl_load_file(lib, "rl_hello.rl");
  lib_function = rl_sym(lib, "_library_function");
  if (!lib_function) {
    printf("Error %d: %s (%s)\n", rl_errno(lib), rl_errstr(lib),
            rl_errarg(lib));
    exit(1);
  }
  lib_function();
  return 0;
}
```

Note: The library requires the `printf` *symbol to be present in the main module. The* `printf` *symbol is naturally present because the main module also requires the* `printf` *symbol. If this is not the case, then take the necessary steps to ensure that it is linked in, otherwise the relocatable library will fail to load.*

11.5.2 Building a simple relocatable library

To build a relocatable library that can be loaded by the **rl_lib** loader, additional compiler and linker options must be used.

To build a loadable module, the `-fpic` option is required when compiling and the `-rlib` option is required when linking.

The following is a simple example of building the `hello world` loadable module:

```
sh4gcc -o rl_hello.o -fpic -c rl_hello.c
sh4gcc -o rl_hello.rl -mruntime=os21 -rlib rl_hello.o
```

Alternatively, the relocatable library can be built using a single command:

```
sh4gcc -o rl_hello.rl -fpic -mruntime=os21 -rlib rl_hello.c
```

11.5.3 Building a simple main module

To build a main module suitable for loading a relocatable library, only the `-rmain` option is required when linking. No special compile time options are required for the main module.

The following is an example of building a main module capable of loading modules for the STb7100-MBoard:

```
sh4gcc -o main.out -mboard=mb411 -mruntime=os21 -rmain main.c
```

Note: This example assumes that the main module provides all the standard library symbols required by the relocatable library.

11.5.4 Running and debugging the main module

A main module can be loaded into GDB as normal, for example:

```
sh4gdb main.out
```

The debugger does not become aware of symbols in the relocatable libraries until the module has loaded them.

For example, given the previous example in *Section 11.5.1* (compiled with the `-g` option), the following command attempts to set a breakpoint on an, as yet, unknown symbol:

```
(gdb) break library_function
Function "library_function" not defined.
Make breakpoint pending on future shared library load? (y or [n])
```

Answering `y` sets the breakpoint on a symbol of that name, if and when one is loaded.

See *Section 11.6 on page 160* for details on how to debug issues with relocatable library loading and linking.

11.5.5 Importing and exporting symbols

For the relocatable loader system to function, the main module (or a loaded module) must provide services to the other load modules. In order to avoid a load error when loading a module, the normal strategy is to link the referenced symbols into the main module (or loaded module). One method of ensuring that the services required by a load module are provided by the main module (or loaded module) is by linking with an import script.

The **sh4rltool** tool can be used to generate a list of symbols in the form of an import or export script from specified input files. The input files are either load modules (relocatable libraries) or text files specifying a list of symbols, formatted with one symbol on each line.

- An import script is generated from a list of symbols specified in one or more text files or from one or more load module files. In the latter case, **sh4rltool** generates an import script from the set of symbols that the load modules require.

  The import script defines the set of symbols that must be linked into a module, so that they are available to any load module that requires them.
- An export script can also be generated to reduce the size of the dynamic symbol table in the main module or load modules. An export script is not mandatory, as all global symbols are exported by default.

  The export script defines the set of symbols (and only these symbols) that must be exported to the other modules through the dynamic symbol table. These symbols are accessible by the load time symbol binding process and by calls to `rl_sym()` and `rl_sym_rec()`.

Note: Specify the `-h` option to ***sh4rltool*** *to display help on its command line options.*

Import scripts

There are two common scenarios where an import script may be generated.

- When the required services are well defined and the list of symbols can be passed to **sh4rltool** in text files.
- When the list of services is not defined, but the load modules are available and can be passed to **sh4rltool**. The **sh4rltool** tool can generate an import script from the set of symbols required by the load modules.

The following example generates an import script `prog_import.ld` from a list of symbols specified in the text file `prog_import.lst`:

```
sh4rltool -i -s -o prog_import.ld prog_import.lst
```

Note: The names contained in the symbol list (`prog_import.lst` in this example) must be the external linkage forms of the symbols to be imported, without the leading underscore that GCC automatically prefixes to identifiers. For C++ language identifiers, this means that the symbol names are their mangled C language equivalent.

The following example generates an import script `prog_import.ld` from a list of load modules, `liba.rl` and `libb.rl`, that may be loaded by the main module:

```
sh4rltool -i -o prog_import.ld liba.rl libb.rl
```

The main module can then be linked using the import script generated in either of the above examples. For example:

```
sh4gcc -mboard=mb411 -mruntime=os21 -rmain
       -o main.out prog_import.ld ...
```

Export scripts

There are two common scenarios where an export script may be generated.

- An export script can be generated at the same time as the import script. This is because the symbols to export are generally the same as those being imported.
- For a load module that has a clearly defined external interface, the export script can be generated from a text file specifying the list of symbols to export.

The following example generates a combined export and import script, `prog_import_export.ld`, from a list of load modules, `liba.rl` and `libb.rl`. The linker can use the script to link the main module so that only the symbols from `liba.rl` and `libb.rl` are imported into the main module and then exported by it.

```
sh4rltool -i -e -o prog_import_export.ld liba.rl libb.rl
sh4gcc -mboard=mb411 -mruntime=os21 -rmain
       -o main.out prog_import_export.ld ...
```

The following example generates an export script, `liba_export.ld`, for a load module, `liba.rl` with a well-defined interface specified in the text file `liba_export.lst`. The linker can use the script to link the load module so that only the defined interface symbols are exported by the load module.

```
sh4rltool -e -s -o liba_export.ld liba_export.lst
sh4gcc -o liba.rl -rlib liba_export.ld ...
```

11.5.6 Optimization options

When compiling a load module with the `-fpic` option, some overhead occurs in the generated code to access functions and data objects. Compiler options and C language extensions can be used to reduce this overhead.

Since relocatable libraries are not subject to symbol preemption, the `-fvisibility=protected` option can be used when generating position independent code (`-fpic`). The `-fvisibility=protected` option enables the inlining of global functions and can be used as a default option for compiling relocatable libraries. For example:

```
sh4gcc -o rl_hello.o -fpic -fvisibility=protected -c rl_hello.c
```

In addition to this option, fine grain visibility can be specified with the `__attribute__((visibility(...))` GNU C extension at the source code level.

For example, if the external interface of a load module is well defined in a header file, the `__attribute__((visibility("protected"))` can be specified for each function of the external interface. To specify that all other defined functions are internal to the load module, specify the `-fvisibility=hidden` compiler option. This combination of options optimize references from the same file to global objects that are not part of the interface.

For detailed information on the visibility specification refer to *Using the GNU Compiler Collection* and to the ELF System V Dynamic Linking ABI.

11.6 Debugging support

This section provides details of the support available for debugging the relocatable libraries.

11.6.1 GDB support

The debugging of dynamically loaded modules is implemented in the same way as for System V dynamic shared objects. The main module debugging information is loaded at the load time of the application. The load modules debugging information is loaded at the load time of the load modules.

To update debugging information, the loader maintains a list of loaded modules together with their file names (which contain the debugging information) and the load address of the module. Each time a new module is loaded the loader calls a special function. GDB sets a breakpoint on this function and traverses the list when this breakpoint occurs to find new loaded modules and load the debugging information.

To find the file that contains the debug information, the loader must know the location of the load module file. This is automatic in the case of `rl_load_file()` as the file name is specified in the interface. For the `rl_load_buffer()` and `rl_load_stream()` functions, the user must set the file name with a call to the `rl_set_file_name()` function.

The following example enables automatic debugging of a load module loaded with `rl_load_buffer()`:

```
{
   int status;
   rl_handle_t *handle = rl_handle_new(rl_this(), 0);
   if (handle == NULL) { /* error */ }
#ifdef DEBUG_ENABLED
   rl_set_filename(handle, load-module);
#endif
   status = rl_load_buffer(handle, module_image);
   if (status == -1) { /* error */ }
   ...
}
```

where *`load-module`* is the name of the load module.

A problem may occur with Flash ROM applications that use relocatable libraries, in which the breakpoint set by GDB to observe when relocatable libraries are loaded and unloaded may be overwritten when the application is relocated. In order to overcome this problem, GDB supports the commands `enable sharedlibrary` and `disable sharedlibrary` to either enable or disable the insertion of this breakpoint. (The breakpoint is enabled by default.) If the command `disable sharedlibrary` is issued, Flash ROM applications with relocatable libraries can be run without problem.

11.6.2 Verbose mode

The **rl_lib** loader library can be configured to print details of its progress at run time. This is done by setting the `RL_VERBOSE` environment variable.

For example, place the following statement in the main module before using the **rl_lib** loader library functions:

```
putenv("RL_VERBOSE=1");
```

11.7 Action callbacks

Action callbacks may be used with a profiling support library, or a user defined package that needs to be informed that a segment has just been loaded or is on the point of being unloaded, by using the user action callback interface.

The following is an example that iterates over the segment list and declares the executable segments to a profiling support library on the loading and unloading of a module.

```
static int segment_profile(rl_handle_t *handle,
                           rl_segment_info_t *info,
                           void *cookie)
{
  rl_action_t action = *((rl_action_t *)cookie);
  const char *file_name = rl_file_name(handle);
  if (file_name != NULL && (info->seg_flags & RL_SEG_EXEC) {
    if (action == RL_ACTION_LOAD) {
      /* Call profiling interface for adding a code region. */
      profiler_add_region(file_name, info->seg_addr,
                          info->seg_size);
    }
    if (action == RL_ACTION_UNLOAD) {
      /* Call profiling interface for removing a code region. */
      profiler_remove_region(file_name, info->seg_addr,
                             info->seg_size);
    }
  }
  return 0;
}
```

```
static int module_profile(rl_handle_t *handle,
                          rl_action_t action,
                          void *cookie)
{
  rl_foreach_segment(handle, segment_profile, (void *)&action);
  return 0;
}

int main(void)
{
  ...
  if (rl_add_action_callback(RL_ACTION_ALL,
                             module_profile, NULL) == -1) {
    fprintf(stderr, "rl_add_action_callback failed\n");
    exit(1);
  }
  ...
  status = rl_load_file(handle, file_name);
  ...
  return 0;
}
```

12 OS21 Trace

The ST40 Micro Toolset supports tracing of the OS21 kernel activity and APIs and also user defined APIs and activities. To trace the OS21 kernel activity and APIs, an application is linked with instrumented versions of the supplied libraries; this instrumentation writes events to a memory buffer allocated on the target.

To assist with tracing the user's application and any user-supplied libraries, the ST40 Micro Toolset provides the tools **os21usertrace** and **os21usertracegen**.

The tool **os21usertrace** accepts a user-supplied definition file, specifying the APIs and events to be traced, and then generates all the output files required to build a version of the application that is instrumented for tracing. The user events are written to the same memory buffer as the OS21 events. See *Section 12.1.1: os21usertrace host tool on page 164*.

The tool **os21usertracegen** accepts an ELF object or executable file and a list of function names and generates a definition file that can be used by **os21usertrace**. See *Section 12.1.3: os21usertracegen host tool on page 168*.

The trace data is extracted by dumping the trace memory buffer to a file on the host. This file is then decoded using the **os21decodetrace** tool. See *Section 12.5: Analyzing the results on page 174*.

Support and visualization of OS21 Trace is provided in STWorkbench. For more information, search for **OS21 System Activity** in the STWorkbench help system.

In addition, the user may control OS21 Trace using GDB commands (see *Section 12.9: GDB commands on page 183* and *Section 12.10: User GDB control commands on page 188*) and by embedding function calls in the application to enable and disable tracing for specific parts of the application (see *Section 12.11: Trace library API on page 192* and *Section 12.13: User trace runtime APIs on page 206*).

12.1 User trace records

User APIs and user defined events are organized into a three tier hierarchy: **group**, **class**, and **name**. For any application, there can be one or more groups, each of which contain one or more classes, and each class can contain one of more names. **name** is either the name of an API that is to be traced, or a reference to a specific event to be traced. The group and class levels are customizable, and should be chosen to reflect the way in which tracing may be applied.

Tracing can be controlled at any of the three levels. For instance, all the APIs and events belonging to a group can be traced as a single entity, or particular classes within a group can be traced individually. The user can control tracing at runtime, either through customized GDB commands (see *Section 12.10: User GDB control commands on page 188*) or by using APIs linked with the application (see *Section 12.13: User trace runtime APIs on page 206*).

12.1.1 os21usertrace host tool

The ST40 Micro Toolset provides the **os21usertrace** tool to help with instrumenting a user application for tracing with OS21 Trace. **os21usertrace** accepts one or more definition files created by the user, and from these it generates a set of output files. These output files consist of:

- a single GDB command script that defines the control commands for STWorkbench and GDB (see *Section 12.10: User GDB control commands on page 188*)
- a single C source and header file containing the implementation of the instrumented user APIs, custom activity APIs and control APIs to be compiled and linked into the application
- a single linker script file containing the linker options for wrapping the user APIs
- a single control file describing the user APIs and activities being traced, for use by the **os21decodetrace** tool

The structure of the definition files is described in *Section 12.1.2: User definition file on page 165*.

The command line for **os21usertrace** is:

```
os21usertrace {option} {definition-file}
```

The command line options are described in *Table 39*. There is a long form and short form alternative for each option.

Table 39. os21usertrace command line options

Option	Description
`--help`	Display help.
`-h`	
`--decode-script` *`file`*	Create the **os21decodetrace** control file (passed to the `-user` option of **os21decodetrace**).
`-d` *`file`*	
`--gdb-script` *`file`*	Create the GDB command script *`file`*.
`-g` *`file`*	
`--link-script` *`file`*	Create the linker script *`file`*.
`-l` *`file`*	
`--wrap-source` *`file`*	Create the C source *`file`*.
`-s` *`file`*	
`--user-prefix` *`name`*	User control name space prefix. Default is `user`.
`-up` *`name`*	
`--user-code-base` *`code`*	User activity and API trace code base. Default is 0.
`-ucb` *`code`*	
`--user-code-script` *`FUNCTION@FUNCTION@FILE`*	This option is reserved for STMicroelectronics use only.
`-ucs` *`FUNCTION@FUNCTION@FILE`*	

Note: *The* `--wrap-source` *option creates both a C source file and its corresponding header file. The header file has the same name as the source file but with the* `.c` *extension replaced with* `.h`*.*

The following example accepts a definition file called `myapp.def`, and generates an **os21decodetrace** control file called `myapp.in`, a GDB command script called `myapp.cmd`, a linker script called `myapp-wrap.ld`, and a C source file called `myapp-wrap.c`. Although the file is not explicitly named on the command line, using the `-s` option also creates a header file for `myapp-wrap.c` called `myapp-wrap.h`.

```
os21usertrace -d myapp.in -g myapp.cmd -l myapp-wrap.ld -s myapp-wrap.c myapp.def
```

12.1.2 User definition file

os21usertrace takes as its input one or more definition files. This file contains details of the user APIs to be traced by OS21 Trace, and the specifications for the custom activity APIs to be created for use by the user application.

The tool **os21usertracegen** can generate a suitable definition file from an ELF object or executable file and a list of function names. See *Section 12.1.3: os21usertracegen host tool on page 168* for more information.

The structure of a definition file, in modified Backus-Naur Form, is as follows:

```
format ::= spec-list

spec-list ::= spec
            | spec-list spec

spec ::= USER-INCLUDE header-spec
       | USER-API group-class-name type-spec-list
       | USER-ACTIVITY group-class-name type-spec-list
       | # comment

header-spec ::= filename
              | <filename>
              | "filename"

type-spec-list ::= type-spec
                 | type-spec-list type-spec

type-spec ::= { type @ format }
            | { type }

group-class-name ::= identifier @ identifier @ identifier
```

where:

- *`filename`* is the name of a header file
- *`identifier`* is a C identifier
- *`type`* is a C type specification, which is either a C basic type (such as `unsigned int`) or a `typedef` defined in an included header file
- *`format`* is the format specification for *`type`*, and is one of the format codes listed in *Table 40 on page 167*
- *`comment`* is a comment
- all text in bold is literal and are not part of the modified BNF syntax
- the **{** and **}** symbols are literal and not part of the modified BNF syntax

In addition:

- a *spec* definition is terminated by the end of a line (and so cannot be split across multiple lines)
- *group-class-name* describes the hierarchy of the API or activity, and always consists of three components, *group*, *class* and *name*. The *group* and *class* components are reflected in the GDB control commands (see *Section 12.10: User GDB control commands on page 188*) and runtime control APIs (see *Section 12.13: User trace runtime APIs on page 206*). The final component, *name* is either the name of the user function being traced, or is a name used to derive the name of a custom activity API (see *Section 12.13.1: User activity control APIs on page 206*).

  A typical example of a *group-class-name* specification is `libc@heap@_malloc_r`, which names the `_malloc_r` API from the class `heap` in the group `libc`.
- `USER-INCLUDE` specifies the name of the include file which defines a *type* referenced by a *type-spec*.

  The **os21usertrace** tool preserves the style of the *header-spec* used in a `USER-INCLUDE` definition when generating the C source file except when the form *filename* is used, which is transformed into the *<filename>* style of *header-spec*.
- `USER-API` specifies the API within the application to be traced, and consists of two parts. The *group-class-name* provides the name of the API, and the *type-spec-list* specifies the prototype for the API.

  The order of elements in the *type-spec-list* is important. The return type for the API is the first *type-spec* specified in the *type-spec-list*. The types of the parameters of the API are specified by the second and subsequent elements in the *type-spec-list*. For example:

  ```
  USER-API libc@heap@_malloc_r {void*@p} {struct _reent*@p} {size_t*@d}
  ```

  indicates that the `_malloc_r()` API returns an object pointer of type `void`, and accepts two parameters, the first being a pointer to a `_reent` structure and the second being a `size_t`.

  If the return type is `void`, or if the API takes no parameters, use the form of `type-spec` with no *format*, that is: {*type*}. For example:

  ```
  USER-API libc@heap@_free_r {void} {struct_reent*@p} {void*@p}
  ```
- `USER-ACTIVITY` specifies the name of the custom activity API to be created by the **os21usertrace** tool. The specification of *type-spec-list* is the same as for `USER-API`, except that it only specifies the type of each parameter for the API as the return type is always `void`.
- if *type* specifies an explicit function pointer type (that is, *type* is not a `typedef`), then a `%s` placeholder for the parameter name must be inserted into the type definition. This is to aid **os21usertrace** in the generation of the C source file. For example, if the parameter type is `int (*)(void)`, then the *type* specification must be `int (*%s)(void).`

Table 40 lists the available *format* codes used by *type-spec*:

Table 40. Format codes

Code	Description
b	8-bit word
B	pointer to 8-bit word(1)
w	16-bit word
W	pointer to 16-bit word(1)
d	32-bit word
D	pointer to 32-bit word(1)
q	64-bit word
Q	pointer to 64 bit word(1)
s	string(2)
p	object pointer
P	function pointer
T	OS21 task_t pointer

1. If a NULL pointer is used with these format codes, the de-referenced value is assumed to be zero.
2. If a NULL pointer is used with a string, an empty string is assumed.

The following restrictions apply.

- Strings are truncated to 255 characters.
- APIs with variable argument lists are not supported. If possible, convert the API into an equivalent form that takes a va_list parameter, and define this in the definition file. For example, replace int TRACE_Print(const char *format, ...) with the following:

```
int TRACE_VPrint(const char *format, va_list args)
{
  ...
}

int TRACE_Print(const char *format, ...)
{
  int result;
  va_list args;

  va_start(args, format);
  result = TRACE_VPrint(format, args);
  va_end(args);

  return result;
}
```

Next, define `TRACE_VPrint` in the definition file as follows:

```
USER-API STAPI@TRACE@TRACE_VPrint {int@d} {const char*@s} {va_list@p}
```

- Non-scalar argument and return types are not supported. If possible, convert the API into an equivalent form taking a reference to the type and define this in the definition file.
- Avoid defining `USER-ACTIVITY` APIs with parameters with a *`type`* that is a derived type (that is, a `typedef`), unless it can be guaranteed that the derived type is declared when the header file (created by the `--wrap-source` option) is included.

Note: The format codes are used by OS21 Trace to decide how to decode and store the return value and arguments of the user and custom activity APIs.

12.1.3 os21usertracegen host tool

The ST40 Micro Toolset provides the **os21usertracegen** tool to generate a definition file for input to the **os21usertrace** tool. See *Section 12.1.2: User definition file on page 165* for details of the definition file format.

os21usertracegen accepts as input an ELF object or executable file (but not a library archive file) and uses the DWARF debug information[a] contained within to generate the definitions required by **os21usertrace**.

The command line for **os21usertracegen** is:

```
os21usertracegen --input | -i file {option | function-name}
```

where *`file`* is an ELF format file and *`function-name`* is the name of a global function. The default is to generate a definition file for all global functions defined by the DWARF debug information in the ELF format file specified by the `--input` option. The set of global functions contributing to the definitions file can be customized by using command line options to only include functions satisfying a specified criteria.

The command line options are described in *Table 41*. There is a long form and short form alternative for each option.

Note: The position of some options within the command line is significant. Also some options can be specified multiple times.

Table 41. os21usertracegen command line options

Option	Description
General options	
`--input file`	Name of the ELF format object or executable input file containing the DWARF debug information.
`-i file`	
`--warn`	Enable warnings. If specified then **os21usertracegen** issues a warning for each function definition that specifies an unsupported parameter or return type.
`-w`	
`--help`	Display help.
`-h`	

[a] GCC creates an ELF format file with DWARF debug information when the `-g` compilation option is specified.

Table 41. os21usertracegen command line options (continued)

Option	Description
Output format options	
`--decl`	Output definitions using their declared types. The default is to use compatible base types; this avoids the need to specify C include files (see `--include`) declaring the types (required when compiling the **os21usertrace** generated source).
`-b`	
`--tag`	Output definitions using C `struct` or `enum` tags as their base types. The default is to use compatible types, `void*` instead of `struct*` and `int` instead of `enum`; this avoids the need to specify C include files (see `--include`) declaring the types (required when compiling the **os21usertrace** generated source).
`-t`	
`--deref`	Output definitions with format codes to de-reference pointer types. The default is not to de-reference pointer types. For example, with this option the type `int*` is output with the format code of `D` instead of the default format code of `p`. Only use this option with functions that are known to reference valid (that is, initialized) pointers and where a de-reference does not have side effects.
`-n`	
`--string`	Output definitions with the `s` format code to decode `char*` types as a NUL (`\0`) terminated strings. Only use this option with functions that are known to reference valid (that is, NUL terminated) strings.
`-s`	
Function match options	
`function-name`	Specifies that only functions that match the name *`function-name`* are to be included in the definitions file. If no *`function-name`* is specified then the default is to match all function names. The interpretation of *`function-name`* is dictated by which of the `--regexp` and `--noregexp` options are in force (see below for details).
`--regexp`	Specifies that the function names following this option contain a regular expression. This is the default. For example, specifying `-x t1` will match functions with the names `t1`, `t10` and `test1`. This option can be specified more than once.
`-x`	
`--noregexp`	Specifies that the function names following this option do not contain a regular expression. For example, specifying `-X t1` will only match the function with the name `t1`. This option can be specified more than once.
`-X`	
`--file` *`regexp`*	Specifies that only functions with a source file name that matches *`regexp`* are included in the definitions file. The default is to match all source file names.
`-f` *`regexp`*	
`--dir` *`regexp`*	Specifies that only functions with a compilation directory name that matches *`regexp`* are included in the definitions file. The default is to match all compilation directories.
`-d` *`regexp`*	

Table 41. os21usertracegen command line options (continued)

Option	Description
Output grouping options	

`--output file`
`-o file`
	Output the generated definitions to *file*. The default is to send the output to the console.
	Note that this option resets the `--group` and `--class` options to their default values and clears the set of C include files specified by the `--include` option (see below for details).
	This option can be specified more than once.

`--group name`
`-g name`
	Specify *name* as the definition group name for the following function names until the next `--group` or `--output` option or to the end of the command line, whichever is the sooner.
	The default is `group_default`. This option can be specified more than once in order to define multiple group names.

`--class name`
`-c name`
	Specify *name* as the definition class name for the following function names until the next `--class` or `--output` option or to the end of the command line, whichever is the sooner.
	The default is `class_default`. This option can be specified more than once in order to define multiple class names.

`--include file`
`-I file`
	Specify the name of a C include file to add to the definitions file. The set of C include files specified by this option is cleared by the next `--output` option.
	Note that *file* can also be specified as `"file"` or `<file>`.
	This option can be specified more than once.

os21usertracegen output file format

os21usertracegen generates an annotated version of the definition file format (see *Section 12.1.2: User definition file on page 165*) where the annotations provide the following additional information:

- the version of the **os21usertracegen** tool
- the name of the ELF format input file from which the definitions file is derived
- for each contributing compilation unit: the locations of the compiled source file (`##compile unit` annotation) and the compilation directory (`##comp_dir` annotation)
- for each matching function name in the compilation unit: the name of the function (`##function` annotation), the function prototype (`##decl` annotation) and an equivalent function prototype specified with compatible base types (`##base_decl` annotation)
- if no function definition can be generated (because its specification is not supported by OS21 Trace) then the reason is included in the `##function` annotation

In addition:

- functions appear in the definitions file in the order that they are defined in the DWARF debug information, not in the order they are specified on the command line
- a function definition can only be defined once in the definitions file

12.1.4 os21usertracegen example

This section shows an example using an ELF executable file to demonstrate the flexibility of the **os21usertracegen** tool.

1. The first step is to link the application (compiled with DWARF debug information enabled) to generate an ELF executable file called `myapp.out`:

   ```
   sh4gcc application-link-options -g -o myapp.out
   ```

2. From the ELF executable file created in step *1.*, match function names that do not start with `DEBUG_write` and have been compiled in a directory ending in `debug`, and output their definitions to the file `myapp-debug-other.def`.

   ```
   os21usertracegen -i myapp.out -o myapp-debug-other.def
     -d 'myapp-directory.*debug$'
     -g debug -c other '^(?!DEBUG_write)'
   ```

   where *`myapp-directory`* is the directory containing the source code for `myapp`.

3. Match function names that start with `DEBUG_write` and have the same compilation directory as in step *2.*, and output their definitions, de-referencing pointer and string types, to the file `myapp-debug-write.def`:

   ```
   os21usertracegen -i myapp.out -o myapp-debug-write.def
     -d 'myapp-directory.*debug$' -n -s
     -g debug -c write '^DEBUG_write'
   ```

4. Match function names that start with `OS_`, `EVENT_` or `TRACE_` that have not been compiled in a directory ending in `debug`, and output their definitions, de-referencing pointer and string types, to the file `myapp-deref.def`:

   ```
   os21usertracegen -i myapp.out -o myapp-deref.def
     -d 'myapp-directory.*(?<!debug)$' -n -s
     -g myapp -c OS '^OS_' -c EVENT '^EVENT_' -c TRACE '^TRACE_'
   ```

5. Match function names that have the same compilation directory as in step *4.* but excluding those that start with `OS_`, `EVENT_` or `TRACE_`, and output their definitions to the file `myapp-other.def`:

   ```
   os21usertracegen -i myapp.out -o myapp-other.def
     -d 'myapp-directory.*(?<!debug)$'
     -g myapp -c other '^(?!(OS_|EVENT_|TRACE_))'
   ```

6. Use **os21usertrace** to process the definition files (generated in steps *2.* to *5.*) to create a C source file, called `myapp-wrap.c`, containing the instrumented functions (as well as the other companion source files):

   ```
   os21usertrace -d myapp.in -g myapp.cmd
     -l myapp-wrap.ld -s myapp-wrap.c
     myapp-debug-other.def
     myapp-debug-write.def
     myapp-other.def
     myapp-deref.def
   ```

7. Next compile the source file generated in step *6.*:

   ```
   sh4gcc -mruntime=os21 -fno-zero-initialized-in-bss(b) -g -c
     myapp-wrap.c
   ```

8. The final step is to re-link the application with trace enabled (see *Section 12.3: Building an application for OS21 Trace* for details):

```
sh4gcc application-link-options -g -o myapp.out
  myapp-wrap.o -trace -trace-api -trace-api-no-time
  -Wl,@myapp-wrap.ld
```

Note: *The use of single quotes (') in the above examples are not required (nor accepted) by the* ***os21usertracegen*** *tool but are present to illustrate the use of quoting to protect the regular expressions from being interpreted by a Unix shell. Under Windows, use double quotes (") instead of single quotes to protect the regular expressions.*

Use of regular expressions

os21usertracegen uses the Perl Compatible Regular Expressions (PCRE) library, which is more powerful and flexible than many other regular expression libraries. This is an open source library (see *www.pcre.org* for details), with many reference and tutorial resources available on the Internet.

12.2 Print a string to the OS21 Trace buffer

It is possible to invoke a `USER-ACTIVITY` function to record that the program has reached a specified point in its execution. It is also possible to print a string to the OS21 Trace buffer with the `OS21_TRACE_PRINT` API. See *os21_trace_print on page 201* for more information.

12.3 Building an application for OS21 Trace

To enable tracing for an application, link it with the appropriate command line options, `-trace` or `-trace -trace-api`.

Note: *Enabling OS21 API tracing also requires OS21 activity tracing to be enabled. Therefore, to enable OS21 API tracing, the* `-trace` *linker command line option must always precede* `-trace-api`*.*

Table 42 lists the **sh4gcc** linker options required to enable the OS21 Trace features.

Table 42. sh4gcc linker options to enable OS21 Trace

sh4gcc options	Description
`-trace`	Initialize OS21 Trace support. Install OS21 callbacks to monitor kernel events. (See the "Callbacks" chapter in the *OS21 User Manual* for more details.) This option uses the default specs file, `os21trace.specs`.
`-trace-specs=`*`file`*	Use *`file`* in place of `os21trace.specs`.

[b] See *Building on page 180* for an explanation about the use of the `-fno-zero-initialized-in-bss` option.

Table 42. sh4gcc linker options to enable OS21 Trace (continued)

sh4gcc options	Description
`-trace-api`	Use in conjunction with `-trace` to initialize OS21 API tracing for all functions in the OS21 API. When this option is used, all public OS21 functions are wrapped using the GNU linker `--wrap` option. The wrapper functions record the parameters and return values of the OS21 APIs into the trace buffer. This option uses the specs file `os21wrap.specs` (which is referenced by the default specs file, `os21trace.specs`).
`-trace-api-`*`class`*	Use in conjunction with `-trace` to initialize OS21 API tracing for all functions in the specified *`class`* of OS21 API[1] . For example: `-trace -trace-api-event` performs tracing only on the OS21 functions that belong to the class `event`.
`-trace-api-no-`*`class`*	Use this option in conjunction with the `-trace-api` option to exclude the specified *`class`* of API from tracing[1]. For example: `-trace -trace-api -trace-api-no-cache` performs tracing on all OS21 functions *except* those that belong to the class `cache`.
`-trace-no-constructor`	Use this option to disable the automatic initialization of the OS21 Trace buffers.
`-trace-no-destructor`	Use this option to disable the OS21 Trace destructors on application exit.

[1] Where *`class`* is one of the following: `cache`, `callback`, `event`, `exception`, `interrupt`, `kernel`, `memory`, `message`, `mutex`, `partition`, `power`, `profile`, `reset`, `semaphore`, `task`, `time` or `vmem`.

12.4 Running the application

By default, an application built for OS21 Trace initially starts with trace logging disabled. To enable tracing of the OS21 kernel and API from GDB, invoke the following commands:

```
source os21trace.cmd
enable_os21_activity_global
enable_os21_api_global
```

Note: The command script `os21trace.cmd` automatically creates two breakpoints. One is on the function that is invoked when the trace buffer is full, and the other is on the function that is invoked when the task information buffer is full.

To enable tracing for user defined APIs and activities, source the GDB command script that was generated by the `--gdb-script` option of **os21usertrace**. In the following example, that file is named `myapp.cmd`. The example also assumes the default prefix of `user`.

```
source myapp.cmd
enable_user_activity_global
enable_user_api_global
```

12.4.1 Trace buffer

The default for the trace buffer is to wrap. This means that when this buffer is full, tracing wraps to the start of the buffer and overwrites the oldest existing events. In this case, the **trace buffer full** breakpoint does not occur. When the buffer wraps, time stamping continues from the previously recorded sample.

Note: The time recorded also includes time spent when profiling is disabled either as a result of an I/O request or because the ST40 is under control of GDB.

With tracing enabled and while the target is running, timestamped events are written to the trace buffer. To access this data, GDB must take control of the target. To do this, either set a breakpoint and wait for the break to match, or stop the target, either with a **Ctrl+C** from within GDB or the **Stop** button in **STWorkbench** or **Insight**.

When GDB has control of the target, extract the trace data by invoking the following GDB command:

```
flush_all_trace_buffers
```

This command extracts data from the task information and trace buffers, writes them to files on the host and then resets the buffers. The following binary files are created:

`os21trace.bin`	This file contains the contents of the trace buffer.
`os21trace.bin.ticks`	OS21 time information (`time_ticks_per_sec` value for the trace timestamps and the absolute time of the last event saved in the trace buffer).
`os21tasktrace.bin`	This file contains the contents of the task information buffer

Section 12.8: Structure of trace binary files on page 181 provides a description of the format for each of these files.

12.5 Analyzing the results

After the OS21 Trace and the task information buffers have been saved on the host, use the decoder tool **os21decodetrace** to convert this data into various output formats for viewing and analysis.

The command line for **os21decodetrace** is:

```
os21decodetrace {option} trace-file
```

The command line options for **os21decodetrace** are described in *Table 43*.

Table 43. os21decodetrace command line options

Option	Description
`-e exe-file`	Optional name of target executable file. Required to obtain the symbolic names of interrupt handlers.
`-n task-trace-file`	Optional name of the task information data file (for example `os21tasktrace.bin`). This file provides the name and other useful information for each task.

Table 43. os21decodetrace command line options (continued)

Option	Description
`-o` *`output-file`*	Optional output file name. The default is to output to the console. When generating trace data files for STWorkbench (using the `-t workbench` option) the files must be given the extension `.osa`. This is to enable them to be opened automatically in STWorkbench. If any other extension is used, the files must be opened in STWorkbench using the “Open with...” option.
`-os21` *`file`*	Optional name of the control file describing the OS21 APIs and traceable activities. Use this option to override the default definition of OS21 APIs and traceable activities. The format of the control file is described in *Section 12.5.2 on page 177*.
`-m` *`mode`*	Use `-m` to modify the format selected by the `-t` option, where *`mode`* is one of the following values. – `details` shows detailed information for each task and interrupt context. This includes the number of trace records associated with each task or interrupt context, and the time spent (in ticks) executing in the task or interrupt context. Task priority and stack location information is provided for each task context. – `metrics` shows timing metrics for each recorded task and interrupt context. The metrics include the number of times a task or interrupt was scheduled or descheduled, and the minimum, maximum and average times that the task or interrupt context was active or inactive. – `zero` includes in the report the tasks and interrupt handlers that have zero time. – `max` is equivalent to specifying `-m details -m metrics -m zero`. – `min` does not show individual task and interrupt handler information. – `simple` uses an alternative time accounting regime that is based upon the context information recorded with each trace record instead of context changes reported by the OS21 activity monitors. This option is most useful when API tracing has been enabled. – `ticks` to output timing information in ticks. – `usecs` to output timing information in real time at microsecond resolution. This is the default. Not all of the *`mode`* modes are applicable to all output formats. See *Section 12.5.1 on page 176* for more information on the usage of this option.
`-t` *`type`*	Optional output format, where *`type`* is one of the following values. – `summary` to display a summary. This is the default. – `workbench` to generate output in a format suitable for STWorkbench. – `text` to display one record per line. The first field is the absolute time. OS21 API trace records also contain the parameters and return value of each function. – `csv` is similar to `text` except that the token separator is a comma. The `-t` option can be followed by an optional `-m` option to modify the format of the output of **os21decodetrace**.

Table 43. os21decodetrace command line options (continued)

Option	Description
`-user` *`file`*	Optional name of the control file describing the user APIs and traceable activities. The format of the control file is described in *Section 12.5.2 on page 177*.
`trace-file`	Trace data file (for example `os21trace.bin`). Note that **os21decodetrace** assumes that the OS21 time information can be found in the file *`trace-file`*`.ticks` (for example `os21trace.bin.ticks`)

12.5.1 Usage of the -m mode option

The various modes of the `-m` option are intended to be used with specific output formats. *Table 44* shows how the `-m` modes can be combined with the different output formats.

Table 44. Permitted combinations of mode and output format

Mode (-m option)	Output format (-t option)			
	summary	workbench	text	csv
`details`	Yes	No	No	No
`metrics`	Yes	Yes	No	No
`zero`	Yes	No	No	No
`max`	Yes	No	No	No
`min`	Yes	No	No	No
`simple`	Yes	No	No	No
`ticks`	Yes	Yes	Yes	Yes
`usecs`	Yes[1]	Yes	Yes	Yes

[1] Displays real time in microseconds, milliseconds, seconds, minutes and hours.

“Yes” indicates that the given mode generates meaningful output when used for the given output format.

“No” indicates that the mode cannot be used for the given output format.

If no `-t` option precedes the `-m` *`mode`* option, **os21decodetrace** assumes `-t summary`.

12.5.2 os21decodetrace control file

The **os21decodetrace** options `-os21` and `-user` both require a control file that describes the APIs and activities being traced. For user-defined APIs and activities, this file is generated by the `--decode-script` option of the **os21usertrace** tool.

The structure of a control file, in modified Backus-Naur Form, is as follows:

```
file-format ::= spec-list

spec-list ::= spec
            | spec-list spec

spec ::= A = code group-class-name parameter-type-spec
       | P = code group-class-name return-type-spec
                parameter-type-spec

return-type-spec ::= format
                   | format type-list

parameter-type-spec ::= format
                      | format type-list

type-list ::= type
            | type-list type
```

where:

- *code* is a number.
- *group-class-name*, *format* and *type* are strings.
- A record of type **A** specifies an activity and a record of type **P** specifies an API.
- *group-class-name* is the same as used in the `USER-API` and `USER-ACTIVITY` specifications, but with all white space removed. See *Section 12.1.2 on page 165*.
- *format* is a concatenation of all the format codes specified for the parameters of an API or activity. It is zero length for a `void` return type or an empty parameter list, in which case a *type-list* is not present.
- *type* is the the same as used in the `USER-API` and `USER-ACTIVITY` specifications, but with all superfluous white space removed. See *Section 12.1.2 on page 165*.

12.6 Examples

12.6.1 OS21 activity and OS21 API trace

A simple example can be built with both OS21 activity and OS21 API trace capabilities by using the following command line:

```
sh4gcc -mboard=platform -mruntime=os21 -trace -trace-api ...
```

The following GDB session connects to a target (using the ST TargetPack `sh4tp` command), loads the program and runs to a breakpoint on `main`. OS21 Trace is enabled, a breakpoint is set on `func` and the target continued. When the target stops the trace data is extracted and written to the host.

```
(gdb) file a.out
(gdb) sh4tp stmc:platform:st40
(gdb) load
(gdb) break main
(gdb) continue

     ... wait for break match on main

(gdb) source os21trace.cmd
(gdb) enable_os21_activity_global
(gdb) enable_os21_api_global
(gdb) break func
(gdb) continue

     ... wait for break match on func

(gdb) flush_all_trace_buffers
```

The following command decodes the generated trace data and displays a summary:

```
os21decodetrace -e a.out -n os21tasktrace.bin os21trace.bin
```

12.6.2 User API and user activity trace

This section provides a simple example of using OS21 Trace to trace APIs and some custom activity events within a user application.

os21usertrace

The first step is to create a definition file for **os21usertrace**. This specifies each of the user API functions and user activity events to trace, using the format described in *Section 12.1.2: User definition file on page 165*.

Figure 27. Example definition file, myapp.def

```
USER-INCLUDE stdlib.h
USER-INCLUDE malloc.h
USER-INCLUDE os21.h
USER-API libc@sys@_sbrk_r {void*@p} {struct _reent*@p} {ptrdiff_t@d}
USER-API libc@heap@_malloc_r {void*@p} {struct _reent*@p} {size_t@d}
USER-API libc@heap@_memalign_r {void*@p} {struct _reent*@p} {size_t@d} {size_t@d}
USER-API libc@heap@_calloc_r {void*@p} {struct _reent*@p} {size_t@d} {size_t@d}
USER-API libc@heap@_realloc_r {void*@p} {struct _reent*@p} {void*@p} {size_t@d}
USER-API libc@heap@_free_r {void} {struct _reent*@p} {void*@p}
USER-ACTIVITY test@esr@esr_signal {unsigned int@d}
USER-ACTIVITY test@isr@isr_signal {unsigned int@d}
USER-ACTIVITY test@task@task_signal {unsigned int@d}
USER-API test@esr@esr_api {const char*@s} {size_t@d}
USER-API test@isr@isr_api {const char*@s} {size_t@d}
USER-API test@task@task_api {const char*@s} [task_t*@T}
```

The example definition file in *Figure 27*, `myapp.def`, specifies:

- several of the C library heap allocation APIs (`_sbrk_r`, `_malloc_r`, `_memalign_r`, `_calloc_r`, `_realloc_r` and `_free_r`)
- three custom activity event API definitions (`esr_signal`, `isr_signal` and `task_signal`)
- three APIs from the user application (`esr_api`, `isr_api` and `task_api`)

Each are defined using an appropriate *`group-class-name`* triplet, and each API has its return value and parameters defined. Several header files are also required, as these define the types referenced by the APIs.

To generate the source files necessary for building the application, run **os21usertrace** with the following command line.

```
os21usertrace -d myapp.in -g myapp.cmd -l myapp-wrap.ld -s myapp-wrap.c myapp.def
```

Building

Use **sh4gcc** to compile the generated C source file, `myapp-wrap.c`:

```
sh4gcc -mruntime=os21 -fno-zero-initialized-in-bss -g -c myapp-wrap.c
```

Warning: The generated C source file must be compiled using the `-fno-zero-initialized-in-bss` option to ensure that the data structures in target memory used by the generated GDB command scripts are correctly initialized when the application is loaded onto the target.

The next step performs the final link of the application with the generated linker script:

```
sh4gcc -mboard=platform -mruntime=os21 -trace ... myapp-wrap.o -Wl,@myapp-wrap.ld
```

Execution

Use GDB to load and run the application. The following GDB session uses the ST TargetPack `sh4tp` command to connect to the platform.

```
(gdb) file a.out
(gdb) sh4tp stmc:platform:st40
(gdb) load
(gdb) break main
(gdb) continue
```

Source the command script `myapp.cmd` in order to use the GDB commands for controlling tracing:

```
(gdb) source myapp.cmd
(gdb) enable_user_activity_global
(gdb) enable_user_api_global
```

When the trace data has been gathered, use `flush_all_trace_buffers` to flush the data to file. Finally, use **os21decodetrace** to decode the trace file.

```
os21decodetrace -e a.out -user myapp.in -n os21tasktrace.bin os21trace.bin
```

*Note: The `-user myapp.in` option is required so that **os21decodetrace** can interpret the data for the user defined APIs and activities.*

12.7 Trace overhead

It should be understood that OS21 Trace is intrusive. The level of intrusiveness depends upon the choice of linker and runtime options. Therefore, take this into consideration when analyzing the trace results, as tracing affects the real time behavior of the application.

The following points are some of the costs to consider when using OS21 Trace.

- The default trace buffer requires 2 Mbytes of heap. Use the variable `os21_trace_constructor_size` to change the size of the buffer.
- The default trace buffer constructor can be disabled using the `-trace-no-constructor` option. The user can then initialize the trace buffer directly using `os21_trace_initialize()`.
- The default task information buffer requires 64 Kbytes of heap. Use the variable `os21_task_trace_constructor_size` to change the size of the buffer.
- The default task information buffer constructor can be disabled using the `-trace-no-constructor` option. The user can then initialize the task information buffer directly using `os21_task_trace_initialize()`.

Note: For more information on the variables and functions named above, see *Section 12.11: Trace library API on page 192*.

- For a representative audio and video decode application that contains 4 Mbytes of code, the approximate increases in code size are as follows:
  - OS21 activity tracing adds 3 Kbytes (0.1% increase)
  - OS21 API tracing adds 17 Kbytes (0.4% increase), including OS21 activity
- For the same representative application, the approximate times to fill the default sized trace buffer (the ST40 is actually 50% idle during the run) are as follows:
  - OS21 activity tracing takes 25 secs
  - OS21 API tracing takes 1.2 secs, including OS21 activity
- For reference, on a 266 MHz ST40-200 series core, consider the following points.
  - One trace event takes approximately 5 microseconds. This impacts an application’s interrupt latency.
  - It takes 300 milliseconds for GDB to extract the trace buffer and then continue the application for full trace coverage.
- The profile of code and data cache utilization is perturbed.

12.8 Structure of trace binary files

As described in *Section 12.4: Running the application on page 173*, the command `flush_all_trace_buffers` outputs the contents of the trace buffer to three binary files. This section describes the internal structure of each of these files.

In the format column in *Table 45*, *Table 46* and *Table 47*:

- `INT8` is an 8-bit unsigned integer
- `INT16` is a 16-bit unsigned integer, little endian format
- `INT32` is a 32-bit unsigned integer, little endian format
- `INT64` is a 64-bit unsigned integer, little endian format

12.8.1 os21trace.bin

This file contains the contents of the trace buffer. It is a sequence of records, where each record has the structure given in *Table 45*.

Table 45. File format of os21trace.bin

Field	Format	Comment
time-stamp	INT32	Delta from previous trace record
context-code	INT8	See os21_context_e in os21trace/tracecodes.h
context	INT32	task_t object pointer or interrupt INTEVT code.
trace-type	INT8	See os21_trace_type_e in os21trace/tracecodes.h
trace-code	INT*n*	*n* is defined by the *code-size* field in the os21trace.bin.ticks format (see *Table 46*). See os21_activity_e and os21_api_e in os21trace/tracecodes.h
options	INT32	The following bits are set to indicate which of the optional fields are included in the record: 0 to 7: number of *arguments* 8: *caller-address* field 9: *frame-address* field
caller-address	INT32	Optional
frame-address	INT32	Optional
arguments	INT32	Optional

12.8.2 os21trace.bin.ticks

This file contains OS21 time information. It consists of the fields described in *Table 46*.

Table 46. File format of os21trace.bin.ticks

Field	Format	Comment
version	INT32	For the current version, this is 0x00000003
code-size	INT32	Size of the *trace-code* field in the os21trace.bin format (see *Table 45*). The valid sizes are 1, 2 or 4 bytes.
tick-rate	INT64	time_ticks_per_sec()
last-time	INT64	time_now() for most recent trace record

12.8.3 os21tasktrace.bin

This file contains the contents of the task information buffer. It is a sequence of records, where each record has the structure given in *Table 47*.

Table 47. File format of os21tasktrace.bin

Field	Format	Comment
handle	INT32	Task task_t object pointer
priority	INT32	Task priority when created
stack-base	INT32	Location of task stack
stack-size	INT32	Size of task stack
task-name	INT8[16]	Task name

12.9 GDB commands

This section lists the OS21 Trace GDB commands accessible when the file os21trace.cmd is sourced within GDB. For more information on a given command, use the GDB command help *command*.

12.9.1 Buffer full action

```
os21_trace_set_mode stop|wrap            (Default mode is wrap)
os21_task_trace_set_mode stop|wrap       (Default mode is stop)
```

If either mode is set to stop, then a breakpoint is enabled to signal when the buffer is full. If set to wrap, this breakpoint is disabled.

If the buffer is not operating in wrap mode, the data is logged into the buffer only while space is available. When the buffer is full, no more logging occurs until the buffer is emptied and reset.

When the **buffer full** breakpoint is raised, a GDB script invokes the appropriate function to flush the buffer and then continues. The function is one of the following:

- for os21_trace_set_mode, the script calls flush_os21_trace_buffer
- for os21_task_trace_set_mode, the script calls flush_os21_task_trace_buffer

This means that the contents of the buffer are automatically extracted when full to provide a complete log. However, the target is stopped for a comparatively long time during each download.

12.9.2 Enable OS21 Trace

```
enable_os21_trace
```

Enable OS21 Trace logging for both OS21 and user trace events. OS21 Trace logging is enabled by default.

```
disable_os21_trace
```

Disable OS21 Trace logging.

```
show_os21_trace
```

Display the status of OS21 Trace logging.

12.9.3 Enable trace control commands

The following GDB commands control the saving of arguments and context information in the trace records for both OS21 and user trace events.

`enable_os21_trace_control` *`control`*

Enable the saving of the information indicated by *`control`*, where *`control`* is one of the following: `save_activity`, `save_api_enter`, `save_api_exit`, `save_activity_args`, `save_api_enter_args`, `save_api_exit_args`, `save_caller_address` or `save_frame_address`.

`disable_os21_trace_control` *`control`*

Disable the saving of information indicated by *`control`*.

`show_os21_trace_control` *`control`*

Shows whether *`control`* is enabled or disabled.

`enable_os21_trace_control_all`

Enable all controls as a single operation.

`disable_os21_trace_control_all`

Disable all controls as a single operation.

`show_os21_trace_control_all`

Display the controls that are enabled or disabled.

12.9.4 Enable OS21 activity

`enable_os21_activity_global`

Enable the logging of OS21 activity types. Disabled by default.

`disable_os21_activity_global`

Disable the logging of OS21 activity types.

`show_os21_activity_global`

Display the logging status of the OS21 activity types.

12.9.5 Enable OS21 API

`enable_os21_api_global`

Enable the logging of OS21 API types. Disabled by default.

`disable_os21_api_global`

Disable the logging of OS21 API types.

`show_os21_api_global`

Display the logging status of the OS21 API types.

12.9.6 Enable OS21 activity event

`show_os21_activity_classes`

Display the OS21 activity event classes. The supported classes are `task`, `interrupt` and `exception`.

`enable_os21_activity_class_all`

Enable the logging of all OS21 activity events in all classes.

`disable_os21_activity_class_all`

Disable the logging of all OS21 activity events in all classes.

`show_os21_activity_class_all`

Display the logging status of all OS21 activity events in all classes.

`enable_os21_activity_class_`*`class`*

Enable the logging of the OS21 activity events in the class *`class`*, where *`class`* is one of the classes listed by `show_os21_activity_classes`.

`disable_os21_activity_class_`*`class`*

Disable the logging of the OS21 activity events in the class *`class`*.

`show_os21_activity_class_`*`class`*

Display the logging status of the OS21 activity events in the class *`class`*.

`enable_os21_activity` *`code`*

Enable the logging of the OS21 activity event specified by *`code`*. All events are enabled by default. The command `show_os21_activity_class_all` lists all valid *`code`* parameters (see *Section 12.9.11: Type and event enables on page 187*).

For an event to be logged, both the event *`code`* and the type (OS21 activity in this case) must be enabled. Disabling the type prevents logging of all the events that belong to that type, although it does not disable them.

`disable_os21_activity` *`code`*

Disable the logging of the OS21 activity event specified by *`code`*.

`show_os21_activity` *`code`*

Display the logging status of the OS21 activity event specified by *`code`*.

12.9.7 Enable OS21 API function

`show_os21_api_classes`

Display the OS21 API classes. The supported classes are `cache`, `callback`, `event`, `exception`, `interrupt`, `kernel`, `memory`, `message`, `mutex`, `partition`, `power`, `profile`, `reset`, `semaphore`, `task`, `time` or `vmem`.

`enable_os21_api_class_all`

Enable logging of all OS21 APIs in all classes.

`disable_os21_api_class_all`

Disable logging of all OS21 APIs in all classes.

`show_os21_api_class_all`

Display logging status of all OS21 APIs in all classes.

`enable_os21_api_class_`*`class`*

Enable logging of the OS21 API in the class *`class`*, where *`class`* is one of classes reported by `show_os21_api_classes`.

`disable_os21_api_class_`*`class`*

Disable logging of the OS21 API in the class *`class`*.

`show_os21_api_class_`*`class`*

Display the logging status of the OS21 API in the class *`class`*.

`enable_os21_api` *`code`*

Enable the logging of the OS21 API specified by *`code`*. All APIs are enabled by default. The command `show_os21_api_class_all` provides the list of valid *`code`* parameters (see *Section 12.9.11: Type and event enables on page 187*).

For an event to be logged, both the API *`code`* and the type (OS21 API in this case) must be enabled. Disabling the type prevents logging of all the events that belong to that type, although it does not disable them.

`disable_os21_api` *`code`*

Disable the logging of the OS21 API specified by *`code`*.

`show_os21_api` *`code`*

Display the logging status of the OS21 API specified by *`code`*.

12.9.8 Enable task information logging

`enable_os21_task_trace`

Enable logging of task information. Take care to ensure that logging is enabled when tasks are created, otherwise **os21decodetrace** and **STWorkbench** are not able to associate task names with trace data. Enabled by default.

`disable_os21_task_trace`

Disable logging of task information.

`show_os21_task_trace`

Display the logging status of task information.

`enable_os21_activity_task_trace`

Enable logging of task information by OS21 activity events (`task_create` and `task_switch`). Enabled by default.

`disable_os21_activity_task_trace`

Disable logging of task information by OS21 activity events.

`show_os21_activity_task_trace`

Display the status of logging task information by OS21 activity events.

12.9.9 Dump buffer to file

```
dump_os21_trace_buffer file [0|1]
dump_os21_task_trace_buffer file [0|1]
```

Dump the contents of the buffer to *`file`*.

The optional second parameter is the buffer reset argument. If 1 (the default) then the buffer is cleared, otherwise it is not reset and the trace data remains intact.

Note: `file` is created the first time that data is written. Subsequent invocations append data to the existing `file`. Take care to always use the same name for the task information buffer as this holds details of all the tasks created by the application.

A file named *`file`*`.ticks` is also created when dumping the trace buffer.

12.9.10 Flush buffers and reset

`flush_os21_trace_buffer`

is equivalent to invoking

```
dump_os21_trace_buffer os21trace.bin
```

`flush_os21_task_trace_buffer`

is equivalent to invoking

```
dump_os21_task_trace_buffer os21tasktrace.bin
```

`flush_all_trace_buffers`

is equivalent to invoking

```
flush_os21_trace_buffer
flush_os21_task_trace_buffer
```

These functions flush the contents of both the trace and task information buffers to predefined file names and then reset the buffers. They write data to the files (if any data is extracted) `os21trace.bin`, `os21trace.bin.ticks` and `os21tasktrace.bin`.

12.9.11 Type and event enables

To support convenient enabling and disabling of related OS21 events with a single operation, the **events** are divided into **classes***;* and **classes** are divided into **types**. A trace event is logged (written into the trace buffer) only if the event itself is enabled as well as its type.

Two types are supported:

- OS21 activity
- OS21 API

For each of these, the following command displays the logging status of the type (see *Section 12.9.4: Enable OS21 activity on page 184* and *Section 12.9.5: Enable OS21 API on page 184*):

show_*type*_global

The following command lists all the classes in a type:

show_*type*_classes

For example:

```
(gdb) show_os21_activity_classes
exception
general
interrupt
task
```

The following command displays the logging status of all the events that belong to a class:

show_*type*_class_*class*

For example, display the logging status of the OS21 APIs in the `time` class with the command:

```
(gdb) show_os21_api_class_time
time_after = enabled
time_minus = enabled
time_now = enabled
time_plus = enabled
time_ticks_per_sec = enabled
```

The following command displays the logging status of a specific event:

```
show_type event
```

For example, display the status of the OS21 API `semaphore_wait` event with the command:

```
(gdb) show_os21_api semaphore_wait
semaphore_wait = enabled
```

The following alternative command displays the logging status of all events for a type:

```
show_type_class_all
```

Each of the show commands has an enable/disable equivalent, except the `show_`*`type`*`_classes` commands. For example:

```
(gdb) disable_os21_activity task_switch
(gdb) disable_os21_activity_class_interrupt
(gdb) show_os21_activity_class_all
excp_enter = enabled
excp_exit = enabled
excp_install = enabled
excp_uninstall = enabled
general_print = enabled
intr_enter = disabled
intr_exit = disabled
intr_install = disabled
intr_uninstall = disabled
task_create = enabled
task_delete = enabled
task_exit = enabled
task_switch = disabled
```

12.10 User GDB control commands

When used with the `--gdb-script` command line option, the tool **os21usertrace** creates a GDB command script that defines a set of GDB commands for controlling the generation of user trace records. These commands are used to show the status of tracing, or to enable or disable tracing for a given group, class or event.

To make these commands available when debugging the application, source the generated command script (see *Section 12.4: Running the application on page 173*).

Note: The element `user` *in the names of the GDB commands listed in the following sections can be changed with the option* `--user-prefix` *of the* ***os21usertrace*** *tool.*

12.10.1 User activity control commands

os21usertrace creates the following commands for controlling the generation of trace records for user activities. Use these commands for enabling or disabling tracing for any group, class or named activity that was specified in the **os21usertrace** definition file.

Note: If no user activities are defined, then none of the following commands are defined.

`show_user_activity_groups`

Display all the user activity trace groups in the application as a simple list.

`enable_user_activity_group_all`
`disable_user_activity_group_all`

Enable or disable the logging of all the activities for all groups.

`show_user_activity_group_all`

Display the logging status of all the activities for all groups.

`show_user_activity_group_`*`group`*`_classes`

Display all the classes of the user trace group *`group`*, where *`group`* is one of the groups listed by `show_user_activity_groups`.

`enable_user_activity_group_`*`group`*`_class_all`
`disable_user_activity_group_`*`group`*`_class_all`

Enable or disable the logging of all the activities for all classes of the user trace group *`group`*.

`show_user_activity_group_`*`group`*`_class_all`

Display the logging status of all the activities for all classes of the user trace group *`group`*.

`enable_user_activity_group_`*`group`*`_class_`*`class`*
`disable_user_activity_group_`*`group`*`_class_`*`class`*

Enable or disable the logging of all the activities within the class *`class`* of the user trace group *`group`*, where *`class`* is one of the classes listed by `show_user_activity_group_`*`group`*`_classes`.

`show_user_activity_group_`*`group`*`_class_`*`class`*

Display the logging status of all the activities within the class *`class`* of the user trace group *`group`*.

`enable_user_activity` *`code`*[c]
`disable_user_activity` *`code`*[c]

Enable or disable the logging of the user activity *`code`*. All activities are enabled by default. The command `show_user_activity_group_all` lists all the valid *`code`* parameters (see *Section 12.9.11: Type and event enables on page 187*).

For an event to be logged, both the activity *`code`* and the type (user activity in this case) must be enabled. Disabling the type prevents logging of all the events that belong to that type, although it does not disable them.

`show_user_activity` *`code`*[c]

Display the logging status of the user activity *`code`*.

[c] These commands are not qualified by class or group since the activity must be unique.

`enable_user_activity_global`
`disable_user_activity_global`

Enable or disable the logging of user activity types. Disabled by default.

`show_user_activity_global`

Display the logging status of user activity types.

12.10.2 User API control commands

os21usertrace creates the following commands for controlling the generation of trace records for user APIs. Use these commands for enabling or disabling tracing for any group, class or named API that was specified in the **os21usertrace** definition file.

Note: If no user APIs are defined, then none of the following commands are defined.

`show_user_api_groups`

Display all the user API trace groups in the application as a simple list.

`enable_user_api_group_all`
`disable_user_api_group_all`

Enable or disable the logging of all the APIs for all groups.

`show_user_api_group_all`

Display the logging status of all the APIs for all groups.

`show_user_api_group_`*`group`*`_classes`

Display all the classes of the user trace group *`group`*, where *`group`* is one of the groups listed by `show_user_api_groups`.

`enable_user_api_group_`*`group`*`_class_all`
`disable_user_api_group_`*`group`*`_class_all`

Enable or disable the logging of all the APIs for all classes of the user trace group *`group`*.

`show_user_api_group_`*`group`*`_class_all`

Display the logging status of all the APIs for all classes of the user trace group *`group`*.

`enable_user_api_group_`*`group`*`_class_`*`class`*
`disable_user_api_group_`*`group`*`_class_`*`class`*

Enable or disable the logging of all the APIs within the class *`class`* of the user trace group *`group`*, where *`class`* is one of the classes reported by `show_user_api_group_`*`group`*`_classes`.

`show_user_api_group_`*`group`*`_class_`*`class`*

Display the status of all the APIs within the class *`class`* of the user trace group *`group`*.

`enable_user_api` *`code`*(d)
`disable_user_api` *`code`*(d)

Enable or disable the logging of the user API specified by *`code`*. All APIs are enabled by default. The command `show_user_api_group_all` lists all the valid *`code`* parameters (see *Section 12.9.11: Type and event enables on page 187*).

For an event to be logged, both the API *`code`* and the type (user API in this case) must be enabled. Disabling the type prevents logging of all the events that belong to that type, although it does not disable them.

`show_user_api` *`code`*[d]

Display the logging status of the user API specified by *`code`*.

`enable_user_api_global`
`disable_user_api_global`

Enable or disable the logging of user API types. Disabled by default.

`show_user_api_global`

Display the logging status of user API types.

12.10.3 Miscellaneous commands

The following GDB command is also created by **os21usertrace**.

`show_user_decode_trace`

Show the location of the associated **os21decodetrace** control file (that is, the argument passed to its `-user` option).

[d] These commands are not qualified by class or group since the API must have global scope and therefore be unique.

12.11 Trace library API

The OS21 Trace library is provided in `libos21trace.a` and its associated header file is `os21trace.h`.

The functions defined by this API are described in the following sections.

os21_trace_initialize

Create a trace buffer

Definition:

```
typedef enum os21_trace_mode_e {
  os21_trace_mode_stop = 1,
  os21_trace_mode_wrap = 2
} os21_trace_mode_e;

void os21_trace_initialize(
  void * data,
  unsigned int size,
  os21_trace_mode_e mode);
```

Arguments:

`data`	The buffer to use.
`size`	The size in bytes of the buffer to create.
`mode`	Buffer full action (stop or wrap).

Returns: Void

Description: This function allocates and initializes a trace buffer specified by the `size` parameter. If `data` is `NULL`, the API returns the current buffer to the heap and allocates a new buffer specified by `size`.

On startup of OS21 Trace, the default constructor invokes this function to create a buffer of size 2 Mbytes (enough for 128k simple records) in `os21_trace_mode_wrap` mode. This default size can be overridden by the user. See *Section 12.12: Variables and APIs that can be overridden on page 205*.

os21_trace_initialize_data Replace an existing trace buffer

Definition:

```
void os21_trace_initialize_data(
  void * data,
  unsigned int size);
```

Arguments:

`data`	The buffer to use.
`size`	The size in bytes of the buffer to create.

Returns: Void

Description: Replace the existing trace buffer with the buffer specified by the `data` and `size` parameters. If `data` is `NULL`, the API returns the current buffer to the heap and allocates a new buffer of the specified `size`.

This function must not be used before `os21_trace_initialize()` has been called.

Note: `os21_trace_initialize_data()` *can be used to clear the trace buffer if* `data` *refers to the existing trace buffer.*

os21_trace_initialize_activity_monitors Initialize activity monitors

Definition:

```
void os21_trace_initialize_activity_monitors(void);
```

Arguments: None

Returns: Void

Description: Use this function to initialize the activity monitors.

os21_trace_set_mode Set the action on trace buffer full

Definition:

```
typedef enum os21_trace_mode_e {
  os21_trace_mode_stop = 1,
  os21_trace_mode_wrap = 2
} os21_trace_mode_e;

os21_trace_mode_e os21_trace_set_mode(os21_trace_mode_e mode);
```

Arguments:

`mode`	Buffer full action (stop or wrap).

Returns: The previous trace mode.

Description: Set the action to be performed when the task trace buffer is full. The options are stop or wrap.

os21_trace_overflow User-defined trace overflow function

Definition:

```
void os21_trace_overflow(
  void * data,
  unsigned int size);
```

Arguments:

`data`	The current trace buffer.
`size`	The size in bytes of data in the buffer.

Returns: Void

Description: A function with this name is called when the trace buffer overflows (in stop mode) or before wraparound occurs (in wrap mode). The `data` and `size` parameters are the current trace data buffer and the size of the data saved in the buffer.

The default implementation of this function is a no-op that the user can override with their own implementation.

os21_task_trace_initialize Create a task information buffer

Definition:

```
void os21_task_trace_initialize(
  void * data,
  unsigned int size,
  os21_task_trace_mode_e mode);
```

Arguments:

`data`	The buffer to use.
`size`	The size in bytes of the buffer to create.
`mode`	Buffer full action (stop or wrap).

Returns: Void

Description: This function allocates and initializes a task information buffer specified by the `size` parameter. If `data` is `NULL`, the API returns the current buffer to the heap and allocates a new buffer specified by `size`.

On startup of OS21 Trace, the default constructor invokes this function to create a buffer of size 64 Kbytes (enough for 2k records) in `os21_trace_mode_wrap` mode. This default size can be overridden by the user. See *Section 12.12: Variables and APIs that can be overridden on page 205*.

os21_task_trace_initialize_data Replace an existing task information buffer

Definition:

```
void os21_task_trace_initialize_data(
  void * data,
  unsigned int size);
```

Arguments:

`data`	The buffer to use.
`size`	The size in bytes of the buffer to create.

Returns: Void

Description: Replace the existing task information buffer with the buffer specified by the `data` and `size` parameters. If `data` is `NULL`, the API returns the current buffer to the heap and allocates a new buffer of the specified `size`.

This function must not be used before `os21_task_trace_initialize()` has been called.

Note: `os21_task_trace_initialize_data()` *can be used to clear the task information buffer if* `data` *refers to the existing task information buffer.*

os21_task_trace_set_mode Set the action on task information buffer full

Definition:

```
typedef enum os21_trace_mode_e {
  os21_trace_mode_stop = 1,
  os21_trace_mode_wrap = 2
} os21_trace_mode_e;

os21_task_trace_mode_e os21_task_ trace_set_mode
  (os21_trace_mode_e mode);
```

Arguments:

`mode`	Buffer full action (stop or wrap).

Returns: The previous trace mode.

Description: Set the action to be performed when the task trace buffer is full. The options are stop or wrap.

os21_task_trace_overflow User-defined task information overflow function

Definition:

```
void os21_task_trace_overflow(
  void * data,
  unsigned int size);
```

Arguments:

`data`	The current task information buffer.
`size`	The size in bytes of data in the buffer.

Returns: Void

Description: A function with this name is called when the task information buffer overflows (in stop mode) or before wraparound occurs (in wrap mode). The `data` and `size` parameters are the current buffer and the size of the data saved in the buffer.

The default implementation of this function is a no-op that the user can override with their own implementation.

os21_trace_set_enable Enable trace logging

Definition:

```
int os21_trace_set_enable(
  int mode);
```

Arguments:

`mode`	Enable (1) or disable (0).

Returns: The previous mode.

Description: Enable or disable OS21 Trace logging. Initially set to 1.

os21_activity_set_global_enable Enable OS21 activity logging

Definition:

```
int os21_activity_set_global_enable(
  int mode);
```

Arguments:

`mode`	Enable (1) or disable (0).

Returns: The previous mode.

Description: Enable or disable OS21 activity logging. Initially set to 0.

os21_activity_set_class_enable Enable OS21 activity logging for class

Definition:

```
typedef enum os21_activity_class_e {
os21_activity_class_exception,
os21_activity_class_interrupt,
os21_activity_class_task,
os21_activity_class_general

os21_activity_class_EOF
} os21_activity_class_e;

void os21_activity_set_class_enable(
  os21_activity_class_e code, int mode);
```

Arguments:

`code`	OS21 activity event class.
`mode`	Enable (1) or disable (0).

Returns: Void

Description: Enable or disable logging for the specified OS21 activity event class.

os21_activity_set_enable Enable OS21 activity logging for activity

Definition:

```
typedef enum os21_activity_e {
os21_activity_task_switch,
os21_activity_task_create,
os21_activity_task_delete,
os21_activity_task_exit,
os21_activity_intr_install,
os21_activity_intr_uninstall,
os21_activity_intr_enter,
os21_activity_intr_exit,
os21_activity_excp_install,
os21_activity_excp_uninstall,
os21_activity_excp_enter,
os21_activity_excp_exit,
os21_activity_general_print,

os21_activity_EOF
} os21_activity_e;

int os21_activity_set_enable(os21_activity_e code, int mode);
```

Arguments:

`code`	OS21 activity event type.
`mode`	Enable (1) or disable (0).

Returns: The previous mode.

Description: Enable or disable logging of the specified OS21 activity event type.

os21_activity_set_task_trace_enable

Enable OS21 task information logging

Definition: `int os21_activity_set_task_trace_enable(int mode);`

Arguments:

`mode`	Enable (1) or disable (0).

Returns: The previous mode.

Description: Enable or disable logging of task information by OS21 activity events (`task_create` or `task_switch`).

os21_api_set_global_enable

Enable OS21 API logging

Definition:

```
int os21_api_set_global_enable(
  int mode);
```

Arguments:

`mode`	Enable (1) or disable (0).

Returns: The previous mode.

Description: Enable or disable OS21 API logging. Initially set to 0.

os21_api_set_class_enable   Enable OS21 API logging for class

Definition:

```
typedef enum os21_api_class_e {
os21_api_class_cache,
os21_api_class_callback,
os21_api_class_event,
os21_api_class_exception,
os21_api_class_interrupt,
os21_api_class_kernel,
os21_api_class_memory,
os21_api_class_message,
os21_api_class_mmap,
os21_api_class_mutex,
os21_api_class_partition,
os21_api_class_power,
os21_api_class_profile,
os21_api_class_reset,
os21_api_class_semaphore,
os21_api_class_scu,
os21_api_class_task,
os21_api_class_time,
os21_api_class_vmem,
os21_api_class_xpu,

os21_api_class_EOF
} os21_api_class_e;

void os21_api_set_class_enable(
  os21_api_class_e code, int mode);
```

Arguments:

`code`	OS21 API class.
`mode`	Enable (1) or disable (0).

Returns: Void

Description: Enable or disable logging for the specified OS21 API class.

os21_api_set_enable — Enable logging for the given API

Definition:

```
int os21_api_set_enable(os21_api_e code, int mode);
```

Arguments:

`code`	OS21 API type.
`mode`	Enable (1) or disable (0).

Returns: The previous mode.

Description: Enable or disable logging of the specified OS21 API type.

os21_task_trace_set_enable — Enable task information logging

Definition:

```
int os21_task_trace_set_enable(
  int mode);
```

Arguments:

`mode`	Enable (1) or disable (0).

Returns: The previous mode.

Description: Enable or disable logging of task information. Initially set to 1.

os21_trace_get_control — Get trace control

Definition:

```
typedef struct os21_trace_control_s {
  unsigned int save_activity:1;
  unsigned int save_api_enter:1;
  unsigned int save_api_exit:1;
  unsigned int save_activity_args:1;
  unsigned int save_api_enter_args:1;
  unsigned int save_api_exit_args:1;
  unsigned int save_caller_address:1;
  unsigned int save_frame_address:1;
} os21_trace_control_t;

void os21_trace_get_control(os21_trace_control_t *control);
```

Arguments:

`control`	The control settings.

Returns: Void

Description: Get the control settings for OS21 Trace.

os21_trace_set_control

Set trace control

Definition:

```
typedef struct os21_trace_control_s {
  unsigned int save_activity:1;
  unsigned int save_api_enter:1;
  unsigned int save_api_exit:1;
  unsigned int save_activity_args:1;
  unsigned int save_api_enter_args:1;
  unsigned int save_api_exit_args:1;
  unsigned int save_caller_address:1;
  unsigned int save_frame_address:1;
} os21_trace_control_t;

void os21_trace_set_control(os21_trace_control_t *control);
```

Arguments:

`control`	The control settings

Returns: Void

Definition: Set the control settings for OS21 Trace.

os21_trace_print

Print a string into the trace buffer

Definition: `void OS21_TRACE_PRINT(const char *string)`

Arguments:

`string`	The string to be written to the buffer

Returns: Void.

Description: Print a string into the trace buffer.

Use the `OS21_TRACE_PRINT()` API in preference to the alternative `os21_trace_print()` API as the former allows the application to link successfully when not linked with the OS21 Trace libraries, whereas the latter does not.

os21_trace_write_file

Write trace buffer to a file

Definition:

```
int os21_trace_write_file(
  const char *name,
  int reset);
```

Arguments:

`name`	File name to create.
`reset`	Clear (1) or keep (0) buffer.

Returns: 0 if OK, 1 if an error occurred.

Description: Write the contents of the trace buffer to the file *`name`*.

The second parameter `reset` is the buffer reset argument. If 1 then the buffer is cleared, otherwise it is not reset and remains intact.

os21_trace_status

Get trace status

Definition:

```
typedef struct os21_trace_status_s {
  int version;
  unsigned int codesize;
  unsigned int size;
  osclock_t tickrate;
  osclock_t lasttime;
} os21_trace_status_t;

void os21_trace_status(os21_trace_status_t *status);
```

Arguments: A structure status with the following fields to be filled in by the function:

version	The version number for the trace buffer format.
codesize	The size of the trace code field in the trace buffer. Valid sizes are 1, 2, or 4 bytes (see *Section 12.8.1: os21trace.bin on page 182*).
size	The current size of the data in the trace buffer.
tickrate	The time_ticks_per_sec value.
lasttime	The time when the last record was logged to the trace buffer.

Returns: Void.

Description: Get the trace buffer status.

os21_trace_write_buffer

Write trace data to memory

Definition:

```
int os21_trace_write_buffer(
  void *data,
  int reset);
```

Arguments:

data	Destination buffer.
reset	Clear (1) or keep (0) buffer.

Returns: 0 if OK, 1 if an error occurred.

Description: Write the contents of the trace buffer to the buffer specified by data. Use os21_task_status() to obtain the size needed for the destination buffer.

The second parameter reset is the buffer reset argument. If 1 then the trace buffer is cleared, otherwise it is not reset and the buffer remains intact.

Use this API in conjunction with os21_trace_status(). To ensure that the information returned by os21_trace_status() remains valid for the call to os21_trace_write_buffer(), these API calls must be encapsulated within calls to os21_trace_set_enable(1) and os21_trace_set_enable(0).

os21_task_trace_write_file — Write task information buffer to a file

Definition:

```
int os21_task_trace_write_file(
  const char *name,
  int reset);
```

Arguments:

`name`	File name to create.
`reset`	Clear (1) or keep (0) buffer.

Returns: 0 if OK, 1 if an error occurred.

Description: Write the contents of the task information buffer to the file *`name`*.

The second parameter `reset` is the buffer reset argument. If 1 then the buffer is cleared, otherwise it is not reset and remains intact.

os21_task_trace_status — Get task information status

Definition:

```
typedef struct os21_task_trace_status_s {
  int version;
  unsigned int size;
} os21_task_trace_status_t;

void os21_task_trace_status(os21_task_trace_status_t *status);
```

Arguments: A structure `status` with the following fields to be filled in by the function:

`version`	The version number for the task information buffer format.
`size`	The current size of the data in the task information buffer.

Returns: Void.

Description: Get the task information buffer status.

os21_task_trace_write_buffer   Write task information data to a buffer

Definition:

```
int os21_task_trace_write_buffer(
  void *data,
  int reset);
```

Arguments:

`data`	Destination buffer.
`reset`	Clear (1) or keep (0) buffer.

Returns: 0 if OK, 1 if an error occurred.

Description: Write the contents of the task information buffer to the buffer specified by `data`. Use `os21_task_trace_status()` to obtain the size needed for the destination buffer.

The second parameter `reset` is the buffer reset argument. If 1 then the buffer is cleared, otherwise it is not reset and remains intact.

Use this API in conjunction with `os21_task_trace_status()`. To ensure that the information returned by `os21_task_trace_status()` remains valid for the call to `os21_task_trace_write_buffer()`, these API calls must be encapsulated within calls to `os21_task_trace_set_enable(1)` and `os21_task_trace_set_enable(0)`.

12.12 Variables and APIs that can be overridden

OS21 Trace provides default constructors for the trace buffer and the task information buffer. The user may customize the constructors for both buffers by overriding the functions and variables listed in this section.

The following variables may be overridden by the user.

```
extern void *os21_trace_constructor_data;
```

Defaults to `NULL`, in which case the initial trace buffer is allocated by `os21_trace_initialize()`. See also *os21_trace_initialize_data on page 193*.

```
extern const unsigned int os21_trace_constructor_size;
```

The size of the trace buffer in bytes. Defaults to 128k records.

```
extern void *os21_task_trace_constructor_data;
```

Defaults to `NULL`, in which case the initial task information buffer is allocated by `os21_task_trace_initialize()`. See also *os21_task_trace_initialize_data on page 195*.

```
extern const unsigned int os21_task_trace_constructor_size;
```

The size of the task information buffer in bytes. Defaults to 2k records.

The following APIs can be overridden by the user.

os21_trace_constructor_user User-definable trace buffer constructor

Definition: `void os21_trace_constructor_user(void);`

Returns: Void.

Definition: The default trace buffer constructor calls a function with this name as its final action. The default implementation of this function is a no-op that the user can override with their own implementation (see *Figure 28 on page 212* for an example).

os21_trace_destructor_user User-definable trace buffer destructor

Definition: `void os21_trace_destructor_user(void);`

Returns: Void.

Definition: The default trace buffer destructor calls a function with this name as its first action. The default implementation of this function is a no-op that the user can override with their own implementation (see *Figure 28 on page 212* for an example).

os21_task_trace_constructor_user — User-definable task information buffer constructor

Definition: `void os21_task_trace_constructor_user(void);`

Returns: Void.

Description: The default task information buffer constructor calls a function with this name as its final action. The default implementation of this function is a no-op that the user can override with their own implementation.

os21_task_trace_destructor_user — User-definable task information buffer destructor

Definition: `void os21_task_trace_destructor_user(void);`

Returns: Void.

Description: The default task information buffer destructor calls a function with this name as its final action. The default implementation of this function is a no-op that the user can override with their own implementation.

12.13 User trace runtime APIs

When used with the `--wrap-source` command line option, the **os21usertrace** tool creates source code that includes a set of APIs that can be called by the application to control the generation of user trace records.

Note: The initial element `user` *in the names of the APIs listed in the following sections can be changed with the option* `--user-prefix` *of the* ***os21usertrace*** *tool.*

12.13.1 User activity control APIs

The following APIs are created by **os21usertrace** for controlling the generation of trace records for custom user activity events.

Note: If no user activities are defined, then none of these APIs are defined.

user_activity_set_group_enable — Enable tracing for an activity group

Definition:

```
void user_activity_set_group_enable(
    user_activity_group_e code, int mode)
```

Arguments:

`code`	Activity group to enable or disable.
`mode`	Enable (1) or disable (0).

Returns: Void.

Description: Enable or disable the logging of all the activities for all classes of the user trace group specified by `code`. The enumeration `user_activity_group_e` is defined in the header file generated by **os21usertrace**.

user_activity_set_group_*group*_class_enable — Enable tracing for an activity class

Definition:

```
void user_activity_set_group_group_class_enable(
     user_activity_group_group_class_e code, int mode)
```

Arguments:

`code`	Activity class to enable or disable.
`mode`	Enable (1) or disable (0).

Returns: Void.

Description: **os21usertrace** generates a set of APIs for enabling or disabling the logging of classes of user defined activities within each of the user defined trace groups. There is one API for each group. For example, if there is a group of user defined activities called `libc`, then the API to enable or disable the logging of any given class of activity within the `libc` group is `user_activity_set_group_libc_class_enable()`.

An enumeration with the name `user_activity_group_`*`group`*`_class_e`, where *`group`* is the name of an activity group, is defined for each activity group in the header file generated by **os21usertrace**.

user_activity_set_enable — Enable tracing for an activity

Definition:

```
int user_activity_set_enable(user_activity_e code, int mode)
```

Arguments:

`code`	Activity to enable or disable.
`mode`	Enable (1) or disable (0).

Returns: 0 for success

Description: Enable or disable the logging of the user defined activity specified by `code`. The enumeration `user_activity_e` is defined in the header file generated by **os21usertrace**.

user_activity_set_global_enable — Enable global tracing for activities

Definition:

```
int user_activity_set_global_enable(int mode)
```

Arguments:

`mode`	Enable (1) or disable (0).

Returns: 0 for success

Description: Enable or disable the logging of user activity types. Initially set to 0.

12.13.2 User API control APIs

The following APIs are created by **os21usertrace** for controlling the generation of trace records for user APIs.

Note: If no user APIs are defined, then none of these APIs are defined.

user_api_set_group_enable — Enable tracing for an API group

Definition:

```
void user_api_set_group_enable(user_api_group_e code, int mode)
```

Arguments:

`code`	API group to enable or disable.
`mode`	Enable (1) or disable (0).

Returns: Void.

Description: Enable or disable the logging of all the APIs for all classes of the user trace group specified by `code`. The enumeration `user_api_group_e` is defined in the header file generated by **os21usertrace**.

user_api_set_group_*group*_class_enable — Enable tracing for an API class

Definition:

```
void user_api_set_group_group_class_enable(
     user_api_group_group_class_e code, int mode)
```

Arguments:

`code`	API class to enable or disable.
`mode`	Enable (1) or disable (0).

Returns: Void.

Description: **os21usertrace** generates a set of APIs for enabling or disabling the logging of classes of user defined APIs within each of the user defined trace groups. There is one API for each group. For example, if there is a group of user defined APIs called `libc`, then the API to enable or disable the logging of any given class of API within the `libc` group is `user_api_set_group_libc_class_enable()`.

An enumeration with the name `user_api_group_`*`group`*`_class_e`, where *group* is the name of an API group, is defined for each API group in the header file generated by **os21usertrace**.

user_api_set_enable

Enable tracing for an API

Definition: `int user_api_set_enable(user_api_e code, int mode)`

Arguments:

`code`	API to enable or disable.
`mode`	Enable (1) or disable (0).

Returns: 0 for success

Description: Enable or disable the logging of the user defined API specified by `code`. The enumeration `user_api_e` is defined in the header file generated by **os21usertrace**.

user_api_set_global_enable

Enable global tracing for APIs

Definition: `int user_api_set_global_enable(int mode)`

Arguments:

`mode`	Enable (1) or disable (0).

Returns: 0 for success

Description: Enable or disable the logging of user API types. Initially set to 0.

12.13.3 User activity APIs

The **os21usertrace** tool creates a set of APIs for generating the user defined events specified in the definition file. These are all named USER_*ACTIVITY*(), where *ACTIVITY* is the name (in upper case letters) of the activity given by the USER-ACTIVITY specification in the definition file (see *Section 12.1.2: User definition file on page 165*). The parameters of the API are determined by the specification given in the definition file.

Note: The preferred version of the API is USER_*ACTIVITY*()*, as this enables the application to be linked successfully even if it is not linked with the OS21 Trace libraries. There is an alternative form of the API, with the name in lower case letters, which does not allow the application to be linked unless it is also linked with the OS21 Trace libraries. Use of the latter API is not recommended.*

12.14 Correspondence between GDB commands and APIs

Table 48 lists the OS21 Trace GDB commands and their equivalent APIs.

Table 48. Correspondence between GDB commands and APIs

GDB command	API
os21_trace_set_mode	os21_trace_set_mode()
os21_task_trace_set_mode	os21_task_trace_set_mode()
enable_os21_trace	os21_trace_set_enable()
disable_os21_trace	
enable_os21_activity_global	os21_activity_set_global_enable()
disable_os21_activity_global	
enable_os21_api_global	os21_api_set_global_enable()
disable_os21_api_global	
enable_os21_activity_class_*class*	os21_activity_set_class_enable()
disable_os21_activity_class_*class*	
enable_os21_activity	os21_activity_set_enable()
disable_os21_activity	
enable_os21_api_class_*class*	os21_api_set_class_enable()
disable_os21_api_class_*class*	
enable_os21_api	os21_api_set_enable()
disable_os21_api	
enable_os21_trace_control	os21_trace_set_control()
enable_os21_trace_control_all	
disable_os21_trace_control	
disable_os21_trace_control_all	
enable_os21_task_trace	os21_task_trace_set_enable()
disable_os21_task_trace	

Table 48. Correspondence between GDB commands and APIs (continued)

GDB command	API
enable_os21_activity_task_trace	os21_activity_set_task_trace_enable()
disable_os21_activity_task_trace	
dump_os21_trace_buffer	os21_trace_write_file()
dump_os21_task_trace_buffer	os21_task_trace_write_file()

Table 49 lists the user GDB commands and their equivalent APIs.

Table 49. Correspondence between GDB commands and APIs

GDB command	API
enable_user_activity_group_all	user_activity_set_group_enable()
disable_user_activity_group_all	
enable_user_activity_group_*group*_class_all	
disable_user_activity_group_*group*_class_all	
enable_user_activity_group_*group*_class_*class*	user_activity_set_group_*group*_class_enable()
disable_user_activity_group_*group*_class_*class*	
enable_user_activity	user_activity_set_enable()
disable_user_activity	
enable_user_activity_global	user_activity_set_global_enable()
disable_user_activity_global	
enable_user_api_group_all	user_api_set_group_enable()
disable_user_api_group_all	
enable_user_api_group_*group*_class_all	
disable_user_api_group_*group*_class_all	
enable_user_api_group_*group*_class_*class*	user_api_set_group_*group*_class_enable()
disable_user_api_group_*group*_class_*class*	
enable_user_api	user_api_set_enable()
disable_user_api	
enable_user_api_global	user_api_set_global_enable()
disable_user_api_global	

12.15 Trace always on

The default is that the OS21 activity and OS21 API logging is disabled at startup. The expectation is that the user enables them using STWorkbench or GDB. However, it may be convenient to always run an application with logging enabled from the outset.

The example in *Figure 28 on page 212* customizes OS21 Trace without having to change the application source. To use this example, compile the example code and add the object to the link command line for the application.

The example defines the following functions.

- `os21_trace_constructor_user()`. This function is called by the trace buffer constructor `os21_trace_constructor` in the OS21 Trace library.
- `os21_trace_destructor_user()`. This function is called by the trace buffer destructor `os21_trace_destructor` in the OS21 Trace library.

Note: The destructor function may not be very useful as embedded applications typically never terminate.

Figure 28. Example to customize trace

```
#include <os21trace.h>

#if !defined(TRACE_SIZE)
#define TRACE_SIZE 256 /* Trace buffer size */
#endif

const unsigned int os21_trace_constructor_size = TRACE_SIZE;

/* Run by OS21 Trace constructor */
void os21_trace_constructor_user(void)
{
  /* Enable trace */
  os21_trace_set_enable(1);
  os21_activity_set_global_enable(1);
  os21_api_set_global_enable(1);
  os21_task_trace_set_enable(1);
}

/* Run by OS21 Trace destructor */
void os21_trace_destructor_user(void)
{
  /* Disable trace */
  os21_trace_set_enable(0);
  os21_task_trace_set_enable(0);

  /* Save trace and task information data */
  os21_trace_write_file("os21trace.bin", 1);
  os21_task_trace_write_file("os21tasktrace.bin", 1);
}
```

12.16 Source directory

The source code for OS21 Trace target library is included with the toolset release. This is located in the `sh-superh-elf/src/os21trace/target` subdirectory of the release installation directory.

There is a `makefile` (GNU `make` compatible) in this directory that has two build rules:

all Build `lib/st40/libos21trace.a` (the default rule).

clean Remove all built files (that is, object files and libraries).

13 Dynamic OS21 profiling

The ST40 Micro Toolset supports profiling using the OS21 profiler under the control of GDB. For this, an application is linked with the dynamic OS21 profiler library. This library enables GDB to control all aspects of the OS21 profiler by sending requests to the application to configure, start and stop the OS21 profiler using standard OS21 APIs. Also, GDB can write the data gathered by the OS21 profiler directly to a file on the host without sending a request to the application. The profile data obtained by GDB can be analyzed using the **os21prof** tool. For more information about OS21 profiling, see the *OS21 user manual* (7358306).

For details of the GDB commands available to control the OS21 profiler, see *Section 13.4: GDB commands on page 215*).

13.1 Overview

The dynamic OS21 profiler adds a monitor task to the application[a]. The purpose of the monitor task is to call OS21 profiler APIs on behalf of GDB.

When GDB needs to call an OS21 profiler API, it writes an action request to a structure in target memory and then raises the OS21 interrupt `OS21_INTERRUPT_HUDI` (reserved for exclusive use by GDB). When the target is restarted, the monitor task is woken up, reads the structure and performs the requested action. On completion of the action, the monitor task writes the result back to the same structure and calls a signal function to inform GDB. GDB can (if configured by the user) read the structure and report the result of the request to the user.

The interface between GDB and the monitor task can be configured by the user. For details of the configuration options, see *Section 13.7.1: Overrides on page 221*.

[a] The monitor task has the name `OS21 Profiler` in the OS21 task list.

13.2 Building an application for dynamic OS21 profiling

Table 50 lists the **sh4gcc** linker options required to enable the dynamic OS21 profiling features.

Table 50. sh4gcc linker options to enable dynamic OS21 profiling

sh4gcc options	Description
`-profiler`	Initialize dynamic OS21 profile support. This option uses the default specs file, `os21profiler.specs`.
`-profiler-specs=`*`file`*	Use *`file`* in place of `os21profiler.specs`.
`-profiler-no-constructor`	Use this option to disable the automatic initialization of the dynamic OS21 profiler. This option also prevents the destructor for the dynamic OS21 profiler from being installed. – The dynamic OS21 profiler constructor is called by the OS21 API `kernel_start()`. – The dynamic OS21 profiler destructor is called during OS21 shutdown.

13.3 Running the application

By default, an application built with dynamic OS21 profiling support initially starts with the OS21 profiler disabled. To enable GDB control of the dynamic OS21 profiler, invoke the following command:

```
source os21profiler.cmd
```

See *Section 13.4: GDB commands on page 215* for a complete list of commands.

13.4 GDB commands

This section lists the dynamic OS21 profiler GDB commands accessible when the file `os21profiler.cmd` is sourced within GDB. For information on a given command, use the GDB command `help` *`command`*.

OS21 profiler initialization

Use the following commands to initialize and de-initialize profiling.

`os21_profiler_initialize` *`instructions-per-bucket frequency`*

Configures the OS21 profiler by calling the OS21 API `profile_init()`. If profiling has already been configured, this command removes the existing configuration by calling `profiler_deinit()` and reconfigures it with the new parameters. If the OS21 profiler is currently running when this command is issued, the OS21 API `profiler_stop()` is called first.

This command accepts two arguments:

- *`instructions-per-bucket`*
  The number of instructions allocated to each bucket when capturing profile data. (A bucket is a counter associated with an address range.)
- *`frequency`*
  Indicates the frequency that samples are to be taken, in hertz.

For example:

```
os21_profiler_initialize 16 5000
```

initializes the OS21 profiler to use 16 instructions per bucket and a sampling frequency of 5 KHz.

`os21_profiler_deinitialize`

Terminates the OS21 profiler by calling the OS21 API `profile_deinit()` to release the memory and resources allocated by `profiler_init()`. If the OS21 profiler is currently running when this command is issued, the OS21 API `profiler_stop()` is called first.

OS21 profiler start

When the OS21 profiler has been initialized, use one of the following commands to start profiling. If the OS21 profiler is already running, the OS21 API `profiler_stop()` is called first.

`os21_profiler_start_all`

Starts the system-wide OS21 profiler (that is, profiling every task and interrupt level) by calling the OS21 API `profile_start_all()`.

`os21_profiler_start_interrupt` *`interrupt-level`*

Starts the OS21 profiler for the specified interrupt level by calling the OS21 API `profile_start_interrupt()`.

`os21_profiler_start_task` *`task-handle`*

Starts the OS21 profiler for the task specified by *`task-handle`* by calling the OS21 API `profile_start_task()`. The argument *`task-handle`* is the address of an OS21 `task_t` object. This address can be extracted from the thread list reported by GDB.

`os21_profiler_start_task_number` *`task-number`*

Starts the OS21 profiler for the task specified by *`task-number`*, where *`task-number`* is the OS21 task number, and not the number assigned to the task by GDB.

This command converts *`task-number`* into a *`task-handle`* by scanning the OS21 task list[b]. The command then calls the OS21 API `profile_start_task()` with *`task-handle`*.

[b] Target memory is read when scanning the task list.

OS21 profiler initialization and start

The OS21 dynamic profiler can be initialized and started using the following combined commands:

```
os21_profiler_initialize_and_start_all instructions-per-bucket frequency
```

This is equivalent to the following:

```
os21_profiler_initialize instructions-per-bucket frequency
os21_profiler_start_all
```

```
os21_profiler_initialize_and_start_interrupt instructions-per-bucket frequency level
```

This is equivalent to the following:

```
os21_profiler_initialize instructions-per-bucket frequency
os21_profiler_start_interrupt level
```

```
os21_profiler_initialize_and_start_task instructions-per-bucket frequency task
```

This is equivalent to the following:

```
os21_profiler_initialize instructions-per-bucket frequency
os21_profiler_start_task task
```

```
os21_profiler_initialize_and_start_task_number instructions-per-bucket frequency task-number
```

This is equivalent to the following:

```
os21_profiler_initialize instructions-per-bucket frequency
os21_profiler_start_task_number task-number
```

OS21 profiler stop

Use the following command to stop profiling.

```
os21_profiler_stop
```

Stops the OS21 profiler by calling the OS21 API `profile_stop()`.

OS21 profiler write data

Use the following commands to write the gathered profile data to a file.

```
os21_profiler_write file
```

Writes the OS21 profile data to *`file`* by calling the OS21 API `profiler_write()`. If the OS21 profiler is currently running, the OS21 API `profiler_stop()` is called first.

```
os21_profiler_dump file
```

Writes the OS21 profile data to *`file`*. The command does not stop the OS21 profiler if it is currently running.

After invoking any of the commands listed above, restart the target to perform the requested action.

Note: The `os21_profiler_dump` *command has immediate effect and therefore the target does not have to be restarted in this case.*

OS21 profiler cancel

Use the following command to cancel a previous command.

`os21_profiler_cancel`

Cancel a previous command if that command is still pending (that is, the dynamic OS21 profiler is not in the `BUSY` state as reported by the `show_os21_profiler_monitor_status` command).

Note: All the commands listed above automatically cancel a previous command if it is still pending, except for `os21_profiler_dump`, which has immediate effect.

OS21 profiler status reporting

Use the following commands to enable or disable the reporting of the status of the OS21 profiler requests.

`enable_os21_profiler_report_signaled`

Enable OS21 profiler request reporting. The target is automatically restarted after reporting the result of the request.

`disable_os21_profiler_report_signaled`

Disable OS21 profiler request reporting.

`enable_os21_profiler_stop_signaled`
`disable_os21_profiler_stop_signaled`

The same as above, but target remains stopped and must be manually restarted.

`show_os21_profiler_monitor_status`

Show the status of the OS21 profiler monitor. *Table 51* lists the possible states.

Table 51. OS21 profiler monitor state

State	Description
`INACTIVE`	The monitor is not initialized.
`IDLE`	The monitor is waiting to perform an action.
`PENDING`	The monitor has yet to start the action.
`BUSY`	The monitor has yet to complete the action.

`show_os21_profiler_status`

Show the status of the OS21 profiler (including the type if active). *Table 52* lists the possible states.

Table 52. OS21 profiler state

State	Description
`INACTIVE`	The profiler is not initialized.
`INITIALIZED`	The profiler is initialized but not started.
`STARTED`(1)	The profiler is running.
`STOPPED`(1)	The profiler is stopped.

1. In this state, profile data is available for dumping by the `os21_profiler_dump` command.

`show_os21_profiler_internal_status`

This is similar to `show_os21_profiler_status` except that it shows the internal status of the OS21 profiler.

13.5 Analyzing the results

After the OS21 profile data has been saved (using the `os21_profile_write` or `os21_profile_dump` commands), use the **os21prof** tool to perform the analysis.

The command line to invoke the **os21prof** tool is as follows:

```
os21prof executable-file profile-file
```

Information on the **os21prof** tool can be found in the *OS21 user manual* (7358306).

13.6 Example

Figure 29 is a listing of an example profiling session controlling the dynamic OS21 profiler with GDB.

Figure 29. Example dynamic OS21 profiler script

```
source os21profiler.cmd
load
#
# Initialise the OS21 profiler (stopping after completion)
#
os21_profiler_initialize 16 5000
enable_os21_profiler_stop_signaled
continue
disable_os21_profiler_stop_signaled
#
# Start system wide profiling and continue until exit
#
os21_profiler_start_all
break exit
continue
#
# Dump the OS21 profile to the host file profile.dat
#
os21_profiler_dump profile.dat
```

13.7 Profiler library API

The dynamic OS21 profiler library is provided in `libos21profiler.a` and its associated header file is `os21profiler.h`.

The functions defined by this API are described in the following sections.

os21_profiler_initialize — Initialize profiling

Definition:

```
typedef struct os21_profiler_init_s {
   size_t instrs_per_bucket;
   int hz;
} os21_profiler_init_t;

int os21_profiler_initialize(
   const os21_profiler_init_t *init);
```

Arguments: A structure `init` with the following fields:

`instrs_per_bucket`	The number of instructions allocated to each bucket.
`hz`	The sampling frequency in hertz.

Returns: `OS21_SUCCESS` for success, or `OS21_FAILURE` if called with invalid parameters, or if out of memory.

Description: Use this function to initialize the dynamic OS21 profiler. If `init` is not `NULL`, then this function calls the OS21 API `profile_init()`, using the contents of the `os21_profiler_init_t` structure. If `init` is `NULL`, then `profile_init()` is not called.

The dynamic OS21 profiler constructor invokes this function with a default initialization parameter of `NULL`. The user can override this default. See *Section 13.7.1: Overrides*.

os21_profiler_deinitialize — Deinitialize profiling

Definition: `int os21_profiler_deinitialize(void);`

Arguments: None

Returns: `OS21_SUCCESS` for success, or `OS21_FAILURE` if the dynamic OS21 profiler cannot be deinitialized.

Definition: Use this function to deinitialize the dynamic OS21 profiler. This function stops the OS21 profiler (if it is running), releases all memory and resources allocated by `os21_profiler_initialize()`.

os21_profiler_monitor_interrupt_clear

User defined interrupt clear function

Description: If the symbol `_os21_profiler_monitor_interrupt` is defined, the dynamic OS21 profiler calls this user defined function to clear the interrupt.

Definition:

```
void os21_profiler_monitor_interrupt_clear(
  interrupt_t *handle);
```

Arguments:

`handle`	The handle of the interrupt to clear.

Returns: None.

os21_profiler_signaled

User defined signal function

Definition: `void os21_profiler_signaled(void);`

Arguments: None

Returns: None

Definition: The dynamic OS21 profiler calls a function with this name when it completes an action requested by the user from GDB.

The default implementation of this function is a no-op that the user can override with their own implementation.

13.7.1 Overrides

Customizing the constructor

The dynamic OS21 profiler provides constructor and destructor functions. The user may customize the constructor by overriding the `os21_profiler_constructor_init` variable.

`os21_profiler_init_t os21_profiler_constructor_init;`

The `init` argument passed to `os21_profiler_initialize()`. If this variable is not defined, `NULL` is passed to the `os21_profiler_initialize()` function.

Configuration of the dynamic OS21 profiler monitor task

The dynamic OS21 profiler uses a dedicated task to monitor for user requests from GDB. See *Section 13.1: Overview on page 214* for details. In the default configuration, GDB uses the ST40 UDI interrupt to signal the monitor task of a pending action. The user may change the interrupt used for signalling the monitor task by overriding the following items.

- Define the symbol:

  `interrupt_name_t _os21_profiler_monitor_interrupt;`

  to specify the interrupt that GDB uses to signal to the monitor task. The default is `OS21_INTERRUPT_HUDI`. This should not normally need changing. This override is defined using the `--defsym` linker option when linking the application, as follows:

```
-Wl,--defsym,_os21_profiler_monitor_interrupt=_OS21_INTERRUPT_name
```

- If the symbol `_os21_profiler_monitor_interrupt` is defined, the dynamic OS21 profiler calls `os21_profiler_monitor_interrupt_clear` to clear the interrupt.
- Define the GDB command `os21_profiler_signal_raise` to raise the interrupt specified by the symbol `_os21_profiler_monitor_interrupt`.

  This command is required only if the interrupt to be raised is not the default (`OS21_INTERRUPT_HUDI`).

If no interrupt is available, the monitor task can be configured to check periodically if an action needs to be performed. The dynamic OS21 profiler provides the following variables to configure this operation.

`unsigned int os21_profiler_monitor_wakeup_period;`

Use this variable to specify the frequency (in hertz) at which the monitor task is to check if an action has been requested. The higher the frequency, the greater the intrusion on the operation of the application. The default is 1 KHz.

`unsigned int os21_profiler_monitor_priority;`

Use this variable to define the priority at which the monitor task runs. By default, this is the maximum OS21 priority (`OS21_MAX_USER_PRIORITY`). It should not be changed unless the monitor task has been configured to periodically check if an action has been requested. Reducing the priority of the monitor task increases the latency between the request being raised and the monitor task performing the action.

Appendix A Toolset tips

This appendix contains miscellaneous tips and advice for using the toolset.

A.1 Managing memory partitions with OS21

OS21 allows memory partitions to be created in order to manage areas of memory. For more information, see the *OS21 User Manual* (7358306). There are several reasons for creating memory partitions, for example:

- to implement an allocation algorithm that is appropriate to an application (for example, to apply some alignment constraint to allocated blocks)
- to manage a special area of memory not visible to the normal memory managers (for example, on-chip RAM or peripheral device RAM)
- to manage a memory region which has special caching issues

To manage a memory partition, do the following.

1. Find the location of the memory and its size. This can be implicitly known; for example, the address and size of on-chip RAM is a characteristic of the SoC.

   To select a pool of memory to manage with an allocator:

   – declare it statically:

   ```
   static unsigned char *my_device_RAM = SOME_ADDRESS;
   ```

   – allocate it from a buffer:

   ```
   static unsigned char my_static_pool[65536];
   ```

   – allocate it from the general heap:

   ```
   unsigned char *my_heap_pool = malloc(65536);
   ```

2. Select the allocation strategy to use with the memory. OS21 has three managers. (See *Section A.2: Memory managers on page 226*.):

   ```
   my_pp = partition_create_simple(my_pool, 65536);
   my_pp = partition_create_fixed(my_pool, 65536, block_size);
   my_pp = partition_create_heap(my_pool, 65536);
   ```

   Alternatively, use a special purpose allocator. To use a special purpose allocator, a partition which uses the required memory management implementation must be created using the `partition_create_any()` API. This API takes the size of a control structure which the allocator uses to manage the memory, and the addresses of functions which perform allocation, freeing, reallocation and status reporting.

The following example implements a simple linear allocator, with no `free` or `realloc` methods.

```
#include <os21.h>
#include <stdio.h>

/*
 * Declare memory to be managed by our partition
 */
static unsigned char my_memory[65536];

/*
 * Declare the management data we use to control the partition
 */
typedef struct {
  unsigned char *base;
  unsigned char *limit;
  unsigned char *free_ptr;
} my_state_t;


/*
 * Allocation routine - really simple!
 */
static void *my_alloc(my_state_t *state, size_t size)
{
  void *ptr = NULL;

  if (size && ((state->free_ptr + size) < state->limit)) {
    ptr = state->free_ptr;
    state->free_ptr = state->free_ptr + size;
  }

  return ptr;
}

/*
 * Partition status routine
 * Note that status->partition_status_used is not filled
 * in here - partition_status sets this field automatically.
 */
static int my_status(my_state_t *state,
                     partition_status_t *status,
                     partition_status_flags_t flag)
{
  status->partition_status_state = partition_status_state_valid;
  status->partition_status_type = partition_status_type_user;
  status->partition_status_size = state->limit - state->base;
  status->partition_status_free = state->limit - state->free_ptr;
  status->partition_status_free_largest =
                     state->limit - state->free_ptr;
}
```

```
/*
 * Initialization routine, called when a partition is created
 */
static void my_initialize(partition_t *pp,
                          unsigned char *base,
                          size_t size)
{
  my_state_t *state = partition_private_state(pp);

  state->free_ptr = base;
  state->base = base;
  state->limit = base + size;
}

int main(void)
{
  partition_t *pp;
  void *ptr;

  /*
   * Start OS21
   */
  kernel_initialize(NULL);
  kernel_start();

  /*
   * Create new partition
   */
  pp = partition_create_any(sizeof(my_state_t),
                            (memory_allocate_fn)my_alloc,
                             NULL, /* no free method */
                             NULL, /* no realloc method */
                            (memory_status_fn)my_status);

  /*
   * Initialize it
   */
  my_initialize(pp, my_memory, sizeof(my_memory));

  /*
   * Try it out!
   */
  printf("Alloc 16 bytes : %p\n", memory_allocate(pp, 16));
  printf("Alloc 10 bytes : %p\n", memory_allocate(pp, 10));
  printf("Alloc  1 bytes : %p\n", memory_allocate(pp,  1));

  printf("Done\n");

  return 0;
}
```

A.2 Memory managers

The run-time libraries provide several memory managers. These provide heap, simple and fixed block allocators. The OS21 heap algorithm is very simple. It maintains a single free list of blocks, and allocates from the first one that can satisfy the request. Blocks added to the free list are coalesced with neighbors to reduce fragmentation.

When OS21 is built with the `-DCONF_DEBUG_ALLOC` option specified, the partition manager in OS21 can provide extensive run-time checking for all partitions, including those maintained by user supplied routines (see *Section A.1 on page 223*).

With the `-DCONF_DEBUG_ALLOC` option enabled, the partition manager over allocates and places scribble guards above and below the block of memory passed back to the user. These guards are filled with a known pattern when the block is allocated, and are checked when the block is freed in order to detect writes which have occurred outside of the block (for example, writing past the end of an array). When OS21 terminates, the partition manager reports any blocks of memory which have been allocated but not freed.

newlib provides Doug Lea's heap memory allocator (version 2.6.4). The design of the allocator is discussed at length in *http://g.oswego.edu/dl/html/malloc.html*. The design goals for this widely used allocator include minimizing execution time and memory fragmentation.

newlib can be rebuilt (see *Section 7.4: Building the packages on page 96*) with debugging switched on in `malloc_r.c` (`-DDEBUG`) to enable extensive run-time checking. With debugging enabled, calls to `malloc_stats()` and `mallinfo()` attempt to check that every memory block in the heap is consistent.

A.3 OS21 scheduler behavior

The scheduler in OS21 provides priority based preemptive FIFO scheduling, with optional timeslicing. The following summarizes its behavior.

- 256 priority levels.
- FIFO scheduling within priority level.
- Tasks are pre-empted when higher priority tasks become runnable.
- Pre-emption can be disabled and re-enabled with `task_lock()` and `task_unlock()`, see *task_lock() and task_unlock() on page 227*.
- Pre-emptions held pending while `task_lock()` is in effect, occur when `task_unlock()` releases the lock.
- Tasks that are pre-empted are placed at the head of the run queue for their priority level.
- Tasks that yield are placed at the tail of the run queue for their priority level.
- Tasks that become runnable are placed at the tail of the run queue for their priority level.
- Tasks that are timesliced are placed at the tail of the run queue for their priority level.
- Timeslicing is optional (off by default), and can be enabled or disabled by calling `kernel_timeslice()`.
- The default timeslice frequency is 50 Hz.
- The timeslice frequency can be set between 1 and 500 Hz by changing the value of the variable `bsp_timeslice_frequency_hz`, either before calling `kernel_initialize()`, or in the BSP library routine `bsp_initialize()`.

A.4 Managing critical sections in OS21

A critical section is a region of code where exactly one thread of execution can run at any one time. There are two forms of critical section to consider:

- task / interrupt
- task / task

A.4.1 task / interrupt critical sections

Within the context of a running task, task / interrupt critical sections are implemented by masking interrupts to prevent the interrupt handler used for serializing from running. OS21 provides three APIs for interrupt masking and unmasking.

interrupt_mask(), interrupt_mask_all() and interrupt_unmask()

These OS21 APIs enables the priority level of the ST40 core to be raised and lowered. The ST40's interrupt level provides a simple mechanism to mask interrupts from signaling the ST40. Any interrupts which have a priority that is strictly greater than the ST40's interrupt priority can interrupt the ST40. Any interrupts which have a priority less than or equal to the ST40's interrupt priority are masked out and cannot therefore affect the core.

The ST40's interrupt level is normally zero, meaning that all interrupts are unmasked. Any interrupt masked by the ST40's interrupt level when it becomes active, is signalled to the ST40 when the ST40's interrupt priority is lowered below that of the active interrupts.

To serialize with an interrupt handler which is interrupting at level *`level`*, only the interrupts up to level *`level`* need to be masked. This stops all interrupts with a priority less than or equal to *`level`* from signalling the ST40, but leaves higher priority interrupts unaffected.

`interrupt_mask()` sets the ST40's interrupt level to the value specified, and `interrupt_mask_all()` sets the ST40's interrupt level to its maximum.

To prevent pre-emption, `interrupt_mask()` and `interrupt_mask_all()` also perform an implicit `task_lock()`. This is because if a context switch occurred whilst under an `interrupt_mask()`, the ST40's interrupt priority would be changed to the value required by the incoming task, thus breaking the critical section. Care should be taken to ensure that an explicit deschedule does not occur whilst interrupts are masked (for example, blocking on a busy semaphore or mutex).

A.4.2 task / task critical sections

OS21 provides a number of mechanisms for achieving task / task critical sections, each of which has its own cost and benefit.

task_lock() and task_unlock()

These calls provide a lightweight mechanism to prevent pre-emption. With a `task_lock()` in effect, the running task is guaranteed that the scheduler will not preempt it if a higher priority task becomes runnable, or a timeslice interrupt occurs. In addition, any calls to `task_reschedule()` have no effect.

It is possible for the running task to explicitly give up the core while a `task_lock()` is active. This is the only way another task can be scheduled while the running task holds a task_lock(). Explicit yielding of the core occurs when the running task calls task_yield() or a blocking OS21 API, for example:

- calls to task_delay() or task_delay_until() specifying a time in the future
- waiting on an unposted event flag, busy semaphore or empty message queue with the timeout period not set to TIMEOUT_IMMEDIATE
- waiting for a busy mutex

When the running task resumes execution, OS21 automatically reinstates task_lock(). Due to the critical section provided by task_lock() and task_unlock() being broken, if the task blocks, it is weak. If a strong critical section is required when using these calls, ensure that called functions do not block. This is not always possible, for example, when calling a library function.

Advantages:

- lightweight
- no need to allocate a synchronization object
- critical sections can nest

Disadvantages:

- critical sections broken if the running task blocks

Mutexes

Mutexes in OS21 provide robust critical sections. The critical section remains in place even if the task in the critical section blocks. Exactly one task is able to own a mutex at any one time. OS21 provides two forms of mutex: FIFO and priority.

FIFO mutexes have the simplest semantics. When tasks try to acquire a busy FIFO mutex they are queued in request order. When a task releases a FIFO mutex, ownership is passed to the task at the head of the waiting queue, and it is unblocked.

Priority mutexes are more complex. When tasks try to acquire a busy priority mutex, they are queued on the mutex in order of descending task priority. In this way, the task at the head of the wait queue is always the one with the highest priority, regardless of when it attempted to acquire the mutex.

Priority mutexes also implement what is known as priority inheritance. This mechanism temporarily boosts the priority of the task that owns a mutex to be the same as the priority of the task at the head of the wait queue. When the owning task releases the mutex, its priority is returned to its original level. This behavior prevents priority inversion, where a low priority task can effectively prevent a high priority task from running. This can happen if a low priority task owns a mutex which a high priority task is waiting for, and a mid level priority task starts running, the low priority task cannot run, and hence cannot release the mutex, causing the high priority task to wait.

Ownership of FIFO or priority mutexes has the effect of making the task immortal, that is, immune to task_kill(). This is intended to prevent deadlock in the event that a task owning a mutex is killed; the mutex would otherwise be left owned by a dead task, and hence it would be locked out for ever. If task_kill() is carried out on a mutex owning task, the task remains running until it releases the mutex, at which point the task_kill() is actioned.

Both forms of mutex can be recursively taken by the owning task without deadlock.

Advantages:

- robust critical section
- can be recursively taken without deadlock
- tasks are immortal while holding a mutex
- FIFO mutexes provide strictly fair access to the mutex
- priority mutexes provide priority ordered access, with priority inheritance

Disadvantages:

- mutexes have to be created before they can be used
- more costly than `task_lock()` and `task_unlock()`
- priority mutexes have a higher cost than FIFO mutexes, due to priority inheritance logic
- strictly for task / task interlock; cannot be used by interrupt handlers

Semaphores

Semaphores in OS21 can be used for a variety of purposes, as discussed in the *OS21 User Manual* (7410372). They can be used to provide a robust critical section, in a similar fashion to mutexes, but with some major differences.

- Semaphores cannot be taken recursively; any attempt to do this results in deadlocking the calling task.
- Like mutexes, both FIFO and priority queuing models are provided, but unlike priority mutexes, priority semaphores do not implement priority inheritance.
- Tasks are not automatically made immortal when they acquire a semaphore.
- Semaphores can be used with care from interrupt handlers.

Advantages

- Robust critical section.
- FIFO and priority queuing models are available, but no priority inheritance.
- No difference in cost between a FIFO and a priority semaphore.
- Due to simpler semantics, slightly lower execution cost compared to mutexes.
- If `TIMEOUT_IMMEDIATE` is used when trying to acquire, and the interrupt handler is written to cope with not acquiring the semaphore, semaphores can be used in an interrupt handler.

Disadvantages

- Semaphores have to be created before they can be used.
- More costly than `task_lock()` and `task_unlock()`.
- Cannot be taken recursively, since the system would deadlock.
- No immortality whilst holding; killing an owning task would be dangerous.

Disabling timeslicing

When running with timeslicing enabled, and a very lightweight task / task critical section is required (which does **not** involve accessing a synchronization object), it is possible to temporarily disable timeslicing. For example:

```
kernel_timeslice(0);
...critical section...
kernel_timeslice(1);
```

Use this approach carefully as the `kernel_timeslice()` API has an immediate global effect. If the task blocks in this region (for example, calls `task_delay()`, blocks waiting for a synchronization object, or signals a synchronization object and gets preempted as a result), then timeslicing remains disabled for all other tasks. This can result in a task not timeslicing in order to share the core.

A.5 Access to uncached memory

OS21 on the ST40 provides the macro functions:

- `ADDRESS_IN_UNCACHED_MEMORY(`*`address`*`)`

  This translates a P1 memory region address (to provide a cached view of physical memory) into the equivalent P2 memory region address (to provide an uncached view of the same physical memory).

- `ADDRESS_IN_CACHED_MEMORY(`*`address`*`)`

  This performs the opposite translation from a P2 to P1 address.

These macro functions only work when an application is executing in non-Space Enhancement (SE) mode. If called from an SE mode application, the macro function panics the OS21 kernel and terminates the application. To avoid this, STMicroelectronics recommend using the virtual memory APIs provided by OS21 to obtain uncached views of physical memory. The advantage of this method is that it is a portable solution that also works with non-SE mode applications.

The following sequence of OS21 virtual memory API calls are equivalent to the macro function `ADDRESS_IN_UNCACHED_MEMORY(`*`address`*`)`.

1. Call `vmem_virt_to_phys()` to obtain the physical address for the virtual address to be accessed through an uncached view.
2. Call `vmem_create()` with the physical address obtained in step *1* with the mode `VMEM_CREATE_UNCACHED|VMEM_CREATE_READ|VMEM_CREATE_WRITE` to obtain a virtual address that gives an uncached view of the physical memory.

   Depending on the mode of the P1 and P2 memory regions, the virtual address returned by `vmem_create()` can be dynamically mapped through the MMU (in the P0 or P3 memory regions) or a static mapping in P1 or P2.
3. Call `vmem_delete()` with the virtual address obtained in step *2* to release the virtual address when it is no longer required.

   If the virtual address has been dynamically mapped via the MMU, use `vmem_delete()` to release a UTLB entry for reuse. This reduces page faults and improves performance.

The following example uses the macro function `ADDRESS_IN_UNCACHED_MEMORY (`*`address`*`)` in a non-SE mode application to obtain an uncached view of a structure in memory that is referenced by the pointer `dev`:

```
struct device *dev, *dev_uc;

dev_uc = (struct device *) ADDRESS_IN_UNCACHED_MEMORY(dev);
```

The following example is the equivalent using the OS21 virtual memory APIs:

```
void *dev_phys;
struct device *dev, *dev_uc;

result = vmem_virt_to_phys(dev, &dev_phys);
assert(result == OS21_SUCCESS);

dev_uc = (struct device *)
       vmem_create(dev_phys, sizeof(struct device), NULL,
       VMEM_CREATE_UNCACHED|VMEM_CREATE_READ|VMEM_CREATE_WRITE);
assert(dev_uc != NULL);

result = vmem_delete(dev_uc);
assert(result == OS21_SUCCESS);
```

A.6 Debugging with OS21

Note: Further information on debugging can be found in the Debugging with GDB manual.

A.6.1 Understanding OS21 stack traces

Every time an application enters OS21 through an interrupt or exception, OS21 captures the context of the core on the current stack. If interrupts nest, it captures multiple contexts, one for each interrupt. The information stored includes the complete register state of the core, details of what caused the context to be saved (interrupt or exception) and the task that was active at the time. More information is reported when the OS21 kernel is built with `CONF_DEBUG` defined (which is the case when the `-mruntime=os21_d` compiler option is used to build the application).

Whenever an unexpected exception occurs, it produces a stack trace. On the ST40, these stack traces have the following general form:

```
OS21: ============================================================
OS21: Stack trace (<n> of <N>)

OS21: Fatal exception detected: ST40 exception code
OS21: Description of exception, possibly with faulting address

* Disassembly of instructions around faulting instruction

+ OS21: Active Task ID     : task ID
+ OS21: Active Task Stackp: stack pointer
+ OS21: Active Task name   : task name

<Register dump>
```

```
OS21: ============================================================
OS21: Stack trace (<n+1> of <N>)

OS21: Took interrupt       : <interrupt EVT code>
+ OS21: Active Task ID     : <task ID>
+ OS21: Active Task Stackp: <stack pointer>
+ OS21: Active Task name   : <task name>

<Register dump> ...
```

Note: The lines marked with `+` *are only shown if the stack frame belongs to a task, not if the stack frame belongs to an interrupt handler; and the line marked with* `*` *is only shown when using an OS21 kernel built with* `CONF_DEBUG`

The first stack trace shows the state of the core at the time the exception occurred. It should be possible to ascertain the cause of the exception from the description of the exception, reported faulting addresses and the register dump.

For example, the program in *Figure 30* creates a task that contains a deliberate misaligned write to memory.

Figure 30. Example program with misaligned write to memory

```
#include <os21.h>

void my_task (void *p)
{
  *((unsigned int *)p) = 0xBA49;
}

int main (void)
{
  task_t *t;

  kernel_initialize(0);
  kernel_start();

  t = task_create(my_task, (void *)0x12344321,
    OS21_DEF_MIN_STACK_SIZE, OS21_MAX_USER_PRIORITY,
    "bang", 0);

  task_wait(&t, 1, TIMEOUT_INFINITY);

  return 0;
}
```

Building this program with the compiler options `-g` and `-mruntime=os21_d`, and running it gives the output shown in *Figure 31*.

Figure 31. Stack trace from example in *Figure 30*.

```
OS21: ============================================================
OS21: Stack trace (1 of 1)

OS21: Fatal exception detected
OS21: Exception Code    : 0x00000100
OS21: Exception Address: 0x12344321
OS21: Exception PC      : 0x8400183C
OS21: Data write to misaligned address

OS21:     0x84001838 mov.l@(15,r1),r1
OS21:     0x8400183A mov.l(3,pc),r2; (0x84001848) 0x0000BA49
OS21: >>> 0x8400183C mov.lr2,@r1
OS21:     0x8400183E add#4,r14
OS21:     0x84001840 movr14,r15
OS21: Active Task ID    : 0x840378D0
OS21: Active Task Stackp: 0x8403B8FC
OS21: Active Task name  : bang

OS21: R0:  0x00000000   R1:   0x12344321  R2:  0x0000BA49
OS21: R3:  0x00000003   R4:   0x12344321  R5:  0x12344321
OS21: R6:  0x00000006   R7:   0x00000007  R8:  0x84001828
OS21: R9:  0x12344321   R10:  0x0000000A  R11: 0x0000000B
OS21: R12: 0x0000000C   R13:  0x0000000D  R14: 0x8403B9DC
OS21: R15: 0x8403B9DC

OS21: FR0_0:  0xFFFFBAD0   FR1_0:   0xFFFFBAD0  FR2_0:  0xFFFFBAD0
OS21: FR3_0:  0xFFFFBAD0   FR4_0:   0xFFFFBAD0  FR5_0:  0xFFFFBAD0
OS21: FR6_0:  0xFFFFBAD0   FR7_0:   0xFFFFBAD0  FR8_0:  0xFFFFBAD0
OS21: FR9_0:  0xFFFFBAD0   FR10_0:  0xFFFFBAD0  FR11_0: 0xFFFFBAD0
OS21: FR12_0: 0xFFFFBAD0   FR13_0:  0xFFFFBAD0  FR14_0: 0xFFFFBAD0
OS21: FR15_0: 0xFFFFBAD0

OS21: FR0_1:  0xFFFFBAD1   FR1_1:   0xFFFFBAD1  FR2_1:  0xFFFFBAD1
OS21: FR3_1:  0xFFFFBAD1   FR4_1:   0xFFFFBAD1  FR5_1:  0xFFFFBAD1
OS21: FR6_1:  0xFFFFBAD1   FR7_1:   0xFFFFBAD1  FR8_1:  0xFFFFBAD1
OS21: FR9_1:  0xFFFFBAD1   FR10_1:  0xFFFFBAD1  FR11_1: 0xFFFFBAD1
OS21: FR12_1: 0xFFFFBAD1   FR13_1:  0xFFFFBAD1  FR14_1: 0xFFFFBAD1
OS21: FR15_1: 0xFFFFBAD1

OS21: FPSCR: 0x000C0000   FPUL:   0xDEADBABE

OS21: GBR:   0x00000000   PC:     0x8400183C   SR:   0x40000000
OS21: MACH:  0xDEADBABE   MACL:   0xDEADBABE   PR:   0x84007B86

OS21: Aborted.
```

The exception has been decoded as a misaligned write to memory, and the bad address is `0x12344321`. It can be seen from the disassembly that the instruction causing the error is:

```
OS21: >>> 0x8400183C mov.l   r2,@r1
```

Looking at the register state for this stack frame we can see that the register R1 contains `0x12344321`, and R2 contains `0x0000BA49` as expected.

A.6.2 Identifying a function that causes an exception

It is not possible to directly identify the function that causes an exception from an OS21 stack trace. However, there are several ways to indirectly establish the function.

Using GDB

To catch the exception in GDB, place a breakpoint on OS21's unexpected exception handler. See *Figure 32*.

Figure 32. Using GDB to find an exception (1)

```
(gdb) b _os21_exception_handler
Breakpoint 1 at 0x8400f18c: file os21/src/os21/exception/exception.c, line 110.
(gdb) c
Continuing.
[Switching to Thread 2147483647]

Breakpoint 1, _os21_exception_handler (exceptionp=0x840349cc)
    at os21/src/os21/exception/exception.c:110
110     {
(gdb) info threads
[New Thread 1]
[New Thread 2]
[New Thread 3]
  Id   Target Id         Frame
  4    Thread 3 ("bang" (active & interrupted) [0x840378d0]) 0x8400183c
    in my_task (p=0x12344321) at test.c:5
  3    Thread 2 ("Idle Task" (active) [0x84034cc0]) _os21_task_launch (
    entry_point=0x8400201c <_scheduler_idle>, datap=0x0) at os21/src/os21/task/task.c:1702
  2    Thread 1 ("Root Task" (active) [0x8402a4e4]) 0x8400b222 in _md_kernel_syscall ()
* 1    Thread 2147483647 ("OS21 System Task" (active & running) [0 (PSEUDO)])
    _os21_exception_handler (exceptionp=0x840349cc) at os21/src/os21/exception/exception.c:110
(gdb)
```

In this example, the thread that hit the breakpoint is a pseudo thread called `OS21 System Task`. This is fabricated by GDB to allow it to present the state of the system.

When the exception occurred, thread 4 is indicated as being interrupted as it was running. To examine the state of this thread, change context to that thread, as shown in *Figure 33*.

Figure 33. Using GDB to find an exception (2)

```
(gdb) thread 4
[Switching to thread 4 (Thread 3)]
#0  0x8400183c in my_task (p=0x12344321) at test.c:5
5         *((unsigned int *)p) = 0xBA49;
(gdb) print /x p
$1 = 0x12344321
(gdb)
```

Using sh4objdump

From the OS21 stack trace (see *Section A.6.1: Understanding OS21 stack traces on page 231*), note the value of the PC register of the first stack trace. In the example above, this is `0x8400183C`. Use **sh4objdump** to generate a disassembly of the program, starting before and stopping after this address. This reveals the name of the function that generated the exception. If it does not, specify an earlier start address. See *Figure 34*.

Figure 34. Using sh4objdump to find an exception

```
sh4objdump -d -j .text --start-address=0x84001828 --stop-address=0x84001848 a.out

a.out:     file format elf32-shl


Disassembly of section .text:

84001828 <_my_task>:
84001828:     e6 2f          mov.l   r14,@-r15
8400182a:     fc 7f          add     #-4,r15
8400182c:     f3 6e          mov     r15,r14
8400182e:     e3 61          mov     r14,r1
84001830:     c4 71          add     #-60,r1
84001832:     4f 11          mov.l   r4,@(60,r1)
84001834:     e3 61          mov     r14,r1
84001836:     c4 71          add     #-60,r1
84001838:     1f 51          mov.l   @(60,r1),r1
8400183a:     03 d2          mov.l   84001848 <_my_task+0x20>,r2! ba49
8400183c:     22 21          mov.l   r2,@r1
8400183e:     04 7e          add     #4,r14
84001840:     e3 6f          mov     r14,r15
84001842:     f6 6e          mov.l   @r15+,r14
84001844:     0b 00          rts
84001846:     09 00          nop
```

The command line options given when running **sh4objdump** determine the nature of the output that the tool generates. For example, the `-d` option given in the above example generates an assembly listing of the machine instructions within the limits specified. For a full list of **sh4objdump** command line options, see the entry for **objdump** in the *GNU Binary Utilities Manual*.

Using sh4addr2line

sh4addr2line provides the source file and line number for a specified address. For example, using the same PC as in *Figure 34* (`0x8400183C`), pass it to **sh4addr2line**, as shown in *Figure 35*.

Figure 35. Using sh4addr2line to find an exception

```
sh4addr2line -e a.out -f 0x8400183C
my_task
source-directory/test.c:5
```

Note: The program must contain debug information.

A.6.3 Catching program termination with GDB

The normal exit path for an application is to call `exit()`, so a breakpoint on this function catches typical application exit scenarios.

However, if OS21 detects an internal error, it panics. When this occurs OS21 calls the function `_os21_kernel_panic()` with a string describing the cause. This function is a good place to set a breakpoint to catch kernel panics. `_os21_kernel_panic()` calls `bsp_panic()`, which provides a hook for a user-defined panic handler.

All exit paths go through the internal run-time library function `_SH_posix_Exit_r()`, so setting a breakpoint here catches every exit path.

A.7 General tips for GDB

This section describes a variety of general tips for GDB.

A.7.1 Handling target connections

To avoid typing a sequence of commands when debugging using the GDB command line interface, use a simple script and invoke it with the `-x` command line option. For example:

```
sh4gdb -x script.cmd
```

To connect to the target, define a user-defined command in the `.shgdbinit` command script. The following example defines a command that connects to a board known as `target1` (in this case an STb7100-MBoard), and loads the program specified, ready for debugging:

```
define target1
  file $arg0
  mb411bypass stmc
  load
end
```

To connect to the target from GDB with the executable `a.out`, invoke `target1` with `a.out` as its parameter:

```
(gdb) target1 a.out
```

A.7.2 Windows path names

Although Windows permits spaces to appear within path names, using spaces may cause some GDB commands to fail. Do not use spaces in path names.

When using autocomplete, GDB does not recognize the usual DOS path name separator, the backslash (`\`). Use the Unix style forward slash (`/`) instead.

Windows permits file names to have 2-byte (wide) characters. Usually, this is not a problem, because although the tools do not understand them, they pass them through and Windows still recognizes them. However, some wide characters contain, as one of their two bytes, the directory separator character ‘`\`’ or ‘`/`’. These are correctly interpreted by Windows, but in some cases are misinterpreted by the GNU tools, leading to malformed paths and apparently missing files and directories.

Note: The preferred encoding for GNU is UTF-8, and there are no problems with 2 (or more) byte unicode encodings being misinterpreted as directory separators.

A.7.3 Debugging OS21 boot from ROM applications

When debugging boot from ROM applications built with the **flasher** and **nandflasher** tools supplied with the Flash ROM examples (see *Chapter 10: Booting OS21 from Flash ROM on page 135*), it is necessary to disable OS21 awareness while debugging the ROM bootstrap. This is necessary because before control is passed to the main application by the ROM bootstrap, the application must be relocated by the bootstrap from Flash ROM to main memory. Until the application is relocated into main memory, the OS21 awareness support in GDB extracts an undefined state from the target, resulting in undefined behavior.

The OS21 awareness in GDB may be enabled and disabled using the `enable rtos` and `disable rtos` GDB commands, respectively (see *Table 17: SuperH configuration specific sh4gdb commands on page 63*). Therefore to safely debug an OS21 boot from ROM application, `disable rtos` should be set until execution has transferred control from the ROM bootstrap to the start of the application (defined by the symbol `_start`) at which point OS21 awareness can be safely enabled using `enable rtos`.

Note:

1. *When debugging boot from ROM applications, it is generally inadvisable to configure the target using the standard connection command as the ROM bootstraps also configure the target, and configuring a target twice is not always reliable. GDB therefore provides the* `sh4` *basic connection commands to connect to a target without configuring it.*
2. *The equivalent when connecting to a target with an ST TargetPack is to specify the* `no_pokes=1` *parameter in the TargetString. (See the chapter called “Standard TargetString parameters” in the ST TargetPack User manual 8020851.)*

The following example illustrates connecting to a target (in this case an STEspresso-Demo board) which has been programmed with a boot from ROM application, and debugging the ROM bootstrap until control is passed to the application.

1. Connect to a target ready to debug a boot from ROM application and disable OS21 awareness.

```
(gdb) file a.out                Main application
(gdb) disable rtos              Disable OS21 awareness
(gdb) sh4 stmc "hardreset"      Connect to target stmc
```

2. Set up the memory mapped registers for the target.

```
(gdb) source registers40.cmd  Define register setup commands
(gdb) source display40.cmd    Define register display commands
(gdb) sti5528_si_regs         Define registers for the target
```

3. Debug target initialization and main application loading. Until execution has transferred into the main application (GDB has jumped to the main application's entrypoint), only the `stepi` and `hbreak` commands can be used for stepping and setting breakpoints. The following can be useful if the executable files for the initialization stages are compiled with debug:

```
(gdb) add-symbol-file elf-file base-address
```

4. Start debugging the main application. It is possible to set software breakpoints in the main application at any time (as normal). However, if the debugger inserts the breakpoint into memory before the ROM bootstrap has completed relocating the application from ROM to main memory, the bootstrap will overwrite and effectively disable the breakpoint. This does not apply to hardware breakpoints. The debugger will reinsert all the software breakpoints the next time the program is interrupted, by a hardware breakpoint, user interrupt (**Ctrl+C**) or any other means. The following procedure demonstrates one way to solve the problem. Re-enable OS21 awareness after hitting the breakpoint in the main application.

```
(gdb) hbreak *&start            Set hardware breakpoint on the
                                first instruction in application

(gdb) continue                  Continue execution until
                                application breakpoint

(gdb) enable rtos               Enable OS21 awareness
```

Note: Only a limited number of hardware breakpoints are available.

Depending on the target platform, different configuration commands may need to be specified to the `sh4` connection command. For example, to connect to an STi5202, STb7100 or STb7109 target connected to an ST Micro Connect 1 or ST Micro Connect Lite without an ST TargetPack, use:

```
(gdb) source stb7100jtag.cmd
(gdb) sh4 stmc "jtagpinout=st40 jtagreset -inicommand stb7100_bypass_setup"
```

A.7.4 Power up and connection sequence

There should be no problem with the order of power up between the host machine and the ST Micro Connect. However, STMicroelectronics recommend that the ST Micro Connect is always powered up before the target board, or that the target is reset after power up of the ST Micro Connect.

The reason for this is that when the ST Micro Connect is unpowered, the ribbon cable connection between the target and the ST Micro Connect can create a situation where the JTAG signals are transiently undefined. If this occurs, a target reset should clear any invalid state.

Note: This applies to all types of ST Micro Connect (STMC1, STMC2 and STMCLite).

A.7.5 Using hardware watchpoints

There are some hardware limitations that occur when using hardware watchpoints, for example, hardware watchpoints have limited capability in the regions they are able to watch. This is described in *Hardware watchpoint support* in *Section B.2.4: Silicon specific commands on page 263*.

Watch expressions can be used with literal addresses instead of symbols (which may require more than one hardware watchpoint to implement). For example, the following watches a 4-Kbyte region at the address `0x84001000` without any access size checking (and no page alignment issues):

```
use-watchpoint-access-size off
watch *(unsigned char[4096] *) 0x84001000
continue
```

GDB supports alternative forms of the watch expression, such as:

```
watch *(unsigned char *) 0x84001000 @ 4096
```

and:

```
watch {unsigned char} 0x84001000 @ 4096
```

A.8 Polling for keyboard input

To enable host keyboard polling from an application running on the target, use the `_SH_posix_PollKey()` function.

_SH_posix_PollKey Enable host keyboard polling

Description: `_SH_posix_PollKey()` polls the host keyboard for a keypress. If no keypress is detected, the function returns `0`. If a keypress is detected then the function returns `1`, and the `int` pointed to by `keycode` receives the ASCII keycode of the key that was pressed.

Definition: `int _SH_posix_PollKey(int *keycode);`

Arguments:

`keycode`	The ASCII keycode of the pressed key.

Returns: `0` if no key was pressed, `1` if a key was pressed.

A.9 Changing ST40 clock speeds using GDB command scripts

Warning: This section describes a method for managing the clocks of ST40 SoCs that predates the introduction of ST TargetPacks. Consequently, this information is *not* compatible with the use of ST TargetPacks.

The ST40 Micro Toolset provides GDB command scripts that define user commands to simplify the programming and display of the clock configuration for the subsystems of ST40 SoCs (for example, the CPU, STBus, memory and peripheral subsystems). See *Table 53 on page 241*.

The general mechanism by which the user commands change the clock frequencies is as follows.

1. Stop the PLL.
2. Set the PLL ratios and setup mode.
3. Restart the PLL.
4. Wait for the PLL to lock.

In general, when changing the internal clock frequencies, it is PLL1 of the primary CLOCKGEN subsystem that is reprogrammed. As a consequence of stopping this PLL, the ST40 reverts to using the external clock as its reference clock frequency. This has the effect of setting the internal clock frequencies to be a ratio of the external clock frequency as defined by the CPG.FRQCR register or the CLOCKGEN.PLL1CR1 register (see the SoC-specific datasheet for further information).

The effect of stopping PLL1 on the internal clocks may break the constraint that the frequency of the ST40 UDI clock (DCK) must be less than the peripheral clock frequency.

Since the default UDI clock frequency adopted by the ST40 Micro Toolset (for an ST Micro Connect 1) is 10 MHz, it is likely that the above constraint will be broken when changing the internal clock frequencies. This is because the standard external clock frequency on STMicroelectronics' boards is 27 MHz and the standard peripheral clock ratios are 1/3, 1/6 and 1/8 of the reference clock frequency, that is, a peripheral clock frequency of 9 MHz, 4.5 MHz or 3.4 MHz respectively.

In order to support the changing of ST40 clock frequencies, the user command `linkspeed` is provided to allow the UDI clock frequency used by GDB to be changed in order to comply with the above constraint.

The `linkspeed` command is invoked as follows:

`linkspeed` *`speed`*`[`*`scale`*`]`

where *`speed`* is a decimal value of the UDI clock frequency and *`scale`* is optional. *`scale`* can be `Hz`, `KHz` or `MHz`. If scale is omitted, `Hz` is assumed.

Permitted frequencies for an ST Micro Connect 1 are:

25MHz	20MHz	12.5MHz	10MHz	6.25MHz	5MHz
3.125MHz	2.5MHz	1.5625MHz	1.25MHz	781.25KHz	625KHz
390.625KHz	312.5KHz	195.312KHz	156.25KHz	97.656KHz	78.125KHz

48.828KHz 39.062KHz 24.414KHz 19.531KHz 12.207KHz 9.765KHz
6.103KHz 4.882KHz 3.051KHz 2.441KHz 1.525KHz 1.220KHz
762Hz 610Hz

If the specified frequency does not equal one of the permitted frequencies, the nearest, lower frequency is used. 610 Hz is the minimum permitted for an ST Micro Connect 1.

The following user commands are defined in `st40clocks.cmd` to set the ST40 clocks to the standard reset mode frequencies (see the relevant SoC datasheet for details):

- `st40_cpu100bus50mem50per25` (mode 0)
- `st40_cpu133bus88mem88per44` (mode 1)
- `st40_cpu150bus100mem100per50` (mode 2)
- `st40_cpu166bus110mem110per55` (mode 3)
- `st40_cpu200bus100mem100per50` (mode 4)
- `st40_cpu250bus125mem125per62` (mode 5)

The following example changes the clock frequencies of a target to mode 3 (assuming that it is currently in mode 0):

```
linkspeed 2.5MHz
st40_cpu166bus110mem110per55
linkspeed 10MHz
```

Where `linkspeed 2.5MHz` changes the UDI clock frequency to below that of the peripheral clock frequency (of 3.4 MHz assuming the target is in mode 0 and a 27 MHz external clock) and `linkspeed 10MHz` returns the UDI clock frequency back to the default.

Table 53 lists:

- GDB command scripts that define the user commands for programming the clocks, see the *ST40 Micro Toolset GDB command scripts* (8045872)
- the user command that displays the clock configuration for each SoC
- examples of SoCs that use this command

Table 53. CLOCKGEN commands

Command script	Display command	Example SoCs
`st40clocks.cmd`	`st40_displayclocks`	ST40RA[1], ST40GX1[1]
`sti5528clocks.cmd`	`sti5528_displayclocks`	STi5528[1]
`stm8000clocks.cmd`	`stm8000_displayclocks`	STm8000[1]
`stb7100clocks.cmd`	`stb7100_displayclocks`	STi5202, STb7100, STb7109
`sti7200clocks.cmd`	`sti7200_displayclocks`	STi7200

[1] Legacy platforms

A.10 Just in time initialization

A common problem when writing a library is performing just in time initialization. It is usually accepted that the first thread to call a library function is responsible for initializing it. This often requires allocating memory or synchronization objects like semaphores. The problem is how to ensure that this is atomic, that is, the initialization is performed precisely once. Allocation can result in the caller blocking; therefore, special consideration has to be given as to how to achieve this atomic initialization.

The following describes a simple strategy which guarantees this atomicity.

For a library to initialize, the first caller must create a semaphore to serialize access to the library resources. The following code, which omits error condition checking to aid clarity, guarantees that the semaphore is created precisely once:

```
static semaphore_t *volatile library_sem = NULL;
...

if (library_sem == NULL)
{
  semaphore_t *local_sem = semaphore_create_fifo(1);
  task_lock();
  if (library_sem == NULL)
  {
    library_sem = local_sem;
  }
  task_unlock();
  if (library_sem != local_sem)
  {
    semaphore_delete(local_sem);
  }
}
```

When this code completes, the library semaphore has been created, if necessary. The first check, which occurs unlocked, is to see if the semaphore already exists. If it does, then there is nothing more to do. If it does not, then the code allocates a new semaphore, but keeps the address of the semaphore in a local variable. If the task is descheduled whilst creating the semaphore, it is possible for another task to enter this routine. It too would see that no library semaphore exists, and would similarly attempt to create a new one. When the task returns from creating the semaphore, it locks the scheduler to prevent preemption. Under this lock it again checks the library semaphore. If it still does not exist, the library semaphore is assigned the address of the semaphore just created. The scheduler is now unlocked.

The lock ensures that precisely one of the competing tasks assigned a non-zero value to the library semaphore pointer. When out of the lock, the library semaphore is checked against the local one. If they are identical, then it is known that the local semaphore was used, and nothing more needs to be done. If they are different, then another task assigned the library semaphore pointer. In this case, the local semaphore must be discarded; it is not needed as the library semaphore already exists.

A.11 Using Cygwin

The ST40 Micro Toolset requires no more than a standard Windows environment for normal operation. However, if a Unix-like environment is desired, the toolset may be used in conjunction with Cygwin (www.cygwin.net).

Cygwin provides a number of Unix-like features to its own applications, but cannot extend this support to other non-Cygwin applications, such as the tools in the ST40 Micro Toolset.

To improve interoperability and to use Cygwin as a build environment, the ST40 Micro Toolset provides a limited amount of support for Cygwin environments.

Many of the tools accept Cygwin paths according to the `ST_CYGPATH_MODE` environment variable, see *Table 54*.

Table 54. ST_CYGPATH_MODE settings

Environment variable setting	Description
`ST_CYGPATH_MODE=off`	No path translation is attempted.
`ST_CYGPATH_MODE=normal` or `ST_CYGPATH_MODE` is not set.	`/cygdrive/X` is converted to `X:/`.(1)
`ST_CYGPATH_MODE=full`	`/cygdrive/X` is converted as above and any other Cygwin mount points (such as `/usr`) are also converted.(1)

1. These modes are supported only when using a version of Cygwin earlier than 1.7. They are not supported when using version 1.7 or later.

There are a few limitations:

- paths must be specified in canonical form (that is, `/cygdrive///c` does not work)
- relative paths cannot pass through these paths (that is, `../../cygdrive/c` does not work)
- Cygwin symbolic links are not understood

The **make** tool provided by the ST40 Micro Toolset does not accept Cygwin paths. Instead use the **make** tool supplied with Cygwin by placing the Cygwin `bin` directory earlier in the `PATH` environment variable than the toolset `bin` directory.

There are a number of other tools provided with the toolset which also do not accept Cygwin paths. In these cases, a Windows path must be specified. To convert a path from Cygwin format to Windows format (and back again), use the Cygwin **cygpath** tool.

The following tools in the ST40 Micro Toolset do not support Cygwin paths:

- **censpect** (see *Section 8.4 on page 110*)
- **os21decodetrace** (see *Section 12.5 on page 174*)
- **os21usertracegen** (see *Section 12.1.3 on page 168*)
- **trcview** (see *Section 8.5 on page 120*)
- GDB `branchtrace` command (see *Section E.2 on page 278*)
- GDB `profiler` command (see *Section D.1 on page 271*)
- GDB `targetpack` command (see *Section F.1 on page 282*)
- ST40 simulator commands (see *Section 8.3 on page 104*)

Cygwin paths are not supported by ST40 applications linked with the **libdtf** I/O interface library (see *Section 1.4.3 on page 24*).

Note: The following tools, implemented as Perl scripts, do support Cygwin paths if they are run using the Perl interpreter supplied by Cygwin:

- **os21prof**
- **os21usertrace**
- **sh4rltool**

A.12 Using precompiled headers

In a large project, it is likely that each source file needs to include the same set of header files. The overhead of the compiler having to process this same set of files for each source file can account for a large proportion of the build time, therefore GCC[a] provides the ability to precompile header files in advance of the normal build operation. Builds that use the precompiled header file instead of normal text header files complete much faster.

To create a precompiled header file, simply compile it in the same mannner as any other source file. If it has the customary extension that identifies it as a C or C++ header file (that is, `.h`, `.hh` or `.H`), GCC creates a precompiled header file from this file automatically. The name of the precompiled header file is the same as the original header file but with a `.gch` extension appended; therefore the compiled version of the input file called *`header`*`.h` is *`header`*`.h.gch`.

If the header file does not have an `.h`, `.hh` or `.H` extension, then use the command line option `-x c-header` or `-x c++-header` (as appropriate) to inform GCC to create a precompiled header file from the input file.

When compiling a source file, whenever GCC encounters a `#include` directive, it searches in the first instance for the precompiled version of the header file before the text version of the file. For example, if `#include "all.h"` appears in the source file, and `all.h.gch` is located in the same directory as `all.h`, then GCC uses the precompiled header file `all.h.gch` in preference to `all.h`, as long as certain conditions are met. If these conditions cannot be met, GCC uses `all.h` instead.

For more information (including details of the conditions that apply to the use of precompiled headers), see the section entitled “Using Precompiled Headers” in *Using the GNU Compiler Collection*.

[a] GCC can be any of the compiler tools, `sh4gcc`, `sh4g++` or `sh4c++`.

Appendix B Development tools reference

This appendix provides a reference for the development tools features that are specific to the ST40 cores.

B.1 Code development tools reference

B.1.1 Preprocessor predefines and asserts

The compiler provides a set of predefined preprocessor macros (built-ins) that are listed in *Table 55*. These are in addition to the standard GNU C Compiler (GCC) predefines (such as defining the version of GCC). For details of these refer to the *Using and Porting the GNU Compiler Collection* manual.

Table 55. Preprocessor predefines and asserts

Predefine or Assert	Compiler option
`cpu=sh` (Assert)[1]	This is always defined.
`machine=sh` (Assert)[1]	This is always defined.
`__BARE_BOARD__`	`-mruntime=bare` [2]
`__BIG_ENDIAN__`	`-mb`
`__LITTLE_ENDIAN__`	`-ml` [2]
`__MOVD__`	`-mfmovd`

`__os21__`
`__OS21_BOARD__`
	`-mruntime=os21`
	`-mruntime=os21_d`

`__sh__`	This is always defined.

`__SH_FPU_ANY__`	`-m4` [2]
	`-m4-single`
	`-m4-single-only`
	`-m4-100`
	`-m4-100-single`
	`-m4-100-single-only`
	`-m4-200`
	`-m4-200-single`
	`-m4-200-single-only`
	`-m4-300`
	`-m4-300-single`
	`-m4-300-single-only`

`__sh3__` [3]
`__SH3__`
	`-m4-nofpu`
	`-m4-100-nofpu`
	`-m4-200-nofpu`
	`-m4-300-nofpu`
	`-m4-400`
	`-m4-500`

Table 55. Preprocessor predefines and asserts (continued)

Predefine or Assert	Compiler option

`__SH4__`	`-m4` [2]
	`-m4-100`
	`-m4-200`
	`-m4-300`

`__SH4_100__`	`-m4-100`
	`-m4-100-single`
	`-m4-100-single-only`
	`-m4-100-nofpu`

`__SH4_200__`	`-m4-200`
	`-m4-200-single`
	`-m4-200-single-only`
	`-m4-200-nofpu`

`__SH4_300__`	`-m4-300`
	`-m4-300-single`
	`-m4-300-single-only`
	`-m4-300-nofpu`

`__SH4_400__`	`-m4-400`
`__SH4_500__`	`-m4-500`

`__SH4_NOFPU__`	`-m4-nofpu`
	`-m4-100-nofpu`
	`-m4-200-nofpu`
	`-m4-300-nofpu`
	`-m4-400`
	`-m4-500`

`__SH4_SINGLE__`	`-m4-single`
	`-m4-100-single`
	`-m4-200-single`
	`-m4-300-single`

`__SH4_SINGLE_ONLY__`	`-m4-single-only`
	`-m4-100-single-only`
	`-m4-200-single-only`
	`-m4-300-single-only`

1. GCC assertions are deprecated, and should not be used.
2. Default option.
3. The ST40 variants without an FPU use the SH-3 ABI (for parameter passing) and define `__SH3__` rather than `__SH4__`. Programs should not confuse the ABI definition with the processor variant as they are not the same.

Note: *All predefined macros in Table 55 can be undefined using the* `-U` *option. For example* `-U__sh__` *undefines* `__sh__` *after it has been defined.*

B.1.2 SH-4 specific GCC options

The GCC options listed in *Table 56* are specific to the SH-4 family of cores (which includes the ST40).

Table 56. SH-4 specific GCC options

Option	Use	Supported
`-m4` [1]	Compile for a generic SH-4 core.	Yes
`-m4-100`	Compile for an ST40-100 series core.	Yes
`-m4-200`	Compile for an ST40-200 series core.	Yes
`-m4-300`	Compile for an ST40-300 series core.	Yes
`-m4-340`	Compile for an ST40-340 series core. All FPU and MMU related instructions are disallowed, including those in assembler inserts. Floating-point calculations are software emulated.	Yes
`-m4-400`	Compile for an ST40-400 series core. All FPU and MMU related instructions are disallowed, including those in assembler inserts. Floating-point calculations are software emulated.	Yes
`-m4-500`	Compile for an ST40-500 series core. All FPU instructions are disallowed, including those in assembler inserts. Floating-point calculations are software emulated.	Yes
`-m4-nofpu`	Compile for a generic SH-4 core with the FPU disabled. No FPU instructions are allowed, including those in assembler inserts. Floating-point calculations are software emulated.	Yes
`-m4-100-nofpu`	This is the same as `-m4-nofpu` but for an ST40-100 series core.	Yes
`-m4-200-nofpu`	This is the same as `-m4-nofpu` but for an ST40-200 series core.	Yes
`-m4-300-nofpu`	This is the same as `-m4-nofpu` but for an ST40-300 series core.	Yes
`-m4-single`	Compile for a generic SH-4 core with a pervading precision of single. The default is double precision.	Yes
`-m4-100-single`	This is the same as `-m4-single` but for an ST40-100 series core.	Yes
`-m4-200-single`	This is the same as `-m4-single` but for an ST40-200 series core.	Yes
`-m4-300-single`	This is the same as `-m4-single` but for an ST40-300 series core.	Yes
`-m4-single-only`	Compile for a generic SH-4 core with double precision disabled. Double precision arithmetic and variables are downgraded to single precision.	Yes
`-m4-100-single-only`	This is the same as `-m4-single-only` but for an ST40-100 series core.	Yes

Table 56. SH-4 specific GCC options (continued)

Option	Use	Supported
`-m4-200-single-only`	This is the same as `-m4-single-only` but for an ST40-200 series core.	Yes
`-m4-300-single-only`	This is the same as `-m4-single-only` but for an ST40-300 series core.	Yes
`-malign-small-blocks=`*`n`*	Set the size in bytes to which branch targets are aligned. The default is n = 16, which is half the size of the cache line. If *n* = 0, align all blocks at the start of a cache line. (This feature only applies to optimization options `-O2` and `-O3`.)	Yes
`-mb`	Generate big endian code.	Yes
`-mbigtable`	Use 4-byte fields for switch tables. This only affects the default field length. When optimizing, the field length is chosen according to actual requirements.	Yes
`-mboard=`*`board`*	Use board support package *`board`*.	Yes
`-mdalign`	Align doubles on 8-byte boundary.	Yes
`-mfmovd`	Use double (8-byte) floating-point loads and enable memory block copying using FPU 64-bit data path. This option assumes `-mdalign` and a change to the ABI.	Yes[2]
`-mhitachi`	Hitachi ABI (Differences in save/restore policy).	No
`-mieee` [3]	Better IEEE conformance (provides NaN and Inf support). Equivalent to `-fno-finite-math-only`.	Yes
`-minline-ic_invalidate`	Inline code to invalidate instruction cache entries after setting up nested function trampolines. Depending on the implementation, this does not always work.	
`-misize`	Annotate assembler listing with estimated address.	Yes
`-ml` [3]	Generate little endian code.	Yes
`-mno-ieee`	Use SH-4 floating-point instructions with no IEEE fixup.	Yes
`-mnomacsave`	Do not save mac registers over function calls.	No
`-mpadstruct`	Pad structs up to multiples of 4 bytes.	Yes
`-mprefergot`	Use `GOT` not `GOTOFF` (PIC code).	Yes
`-mrelax`	Use linker relaxation.	Yes
`-mruntime=`*`runtime`*	Use the run-time library *`runtime`*.	Yes
`-musermode`	Do not generate supervisor-mode only code. If the inline code does not work in user mode, this implies `-mno-inline-ic_invalidate`.	Yes

1. Default option (see *Impact of the -m4 option on the assembler*)
2. Supported for ST40-300 series cores only
3. Default option

Impact of the -m4 option on the assembler

If an `-m4` option is specified with an ST40 core variant (such as `-m4-200` or `-m4-300`), GCC instructs the assembler to reject any instructions that are not supported by that core variant by passing a core specific `--isa=`*`isa`* option to the assembler (such as `--isa=sh4` or `--isa=st40-300).`

If an `-m4` option without an ST40 core variant is specified (such as `-m4`, `-m4-single`, `-m4-single-only` or `-m4-nofpu)`, GCC does not generate instructions specific to a particular core variant (for example, the ICBI instruction of the ST40-300 core). However GCC does instruct the assembler to accept core specific instructions without error by passing the `--isa=sh4-up` or `--isa=sh4-nofpu-up` option to the assembler.

This enables assembly code to be used in generic code that dynamically detects the core variant, and then uses instructions most appropriate for the core type.

Compiling libraries for a specific core

The standard libraries have been compiled for a generic SH-4 core using the options `-m4 -m4-nofpu -m4-single -m4-single-only`. This ensures that these libraries do not contain any core variant specific instructions and can therefore be linked into applications that run on any of the ST40 cores.

Although compiling for a specific core may provide improvements in performance, it also reduces the compatibility of the object files. Object files compiled for a specific variant can only be linked into applications targeted for that specific variant of the ST40 core.

B.1.3 GCC assembler inserts

GCC enables assembler code to be embedded in C functions by using the `asm` statement extension. The format for this statement is:

```
asm ("code" [: [outputs] [: [inputs] [: clobbers]]] )
```

where *`code`* is the assembler code to be inserted in the output file.

The remaining parameters describe the effects of *`code`* on the machine and program state. *`code`* uses `printf`-style parameter names to refer to the parameters that are listed in the *`inputs`* and *`outputs`* sections, for example, `%0` refers to the first value, and `%1` refers to the second value. *`outputs`* is the list of objects modified by *`code`*. *`inputs`* is the list of objects read by *`code`*. *`clobbers`* is the list of values that are modified by *`code`*, and are not listed in the *`outputs`* section. For further details (in general, and for the constraints and qualifiers in particular), refer to the section “Assembler Instructions with C Expression Operands” in *Using and Porting the GNU Compiler Collection* manual.

Each operand is of the form *`qualifier`*`(`*`operand`*`)`. For example:

```
asm ("mov %1, %0" : "=r"(i) : "r"(j)); /* i=j; No clobbers */
```

The set of qualifiers (operand constraints) pertaining to the ST40 are listed in *Table 57*. The `=` means that this operand is write-only for this instruction, the previous value is discarded and replaced by output data. Any letters that are not listed correspond to “no register”. The following is an example of the use of these qualifiers:

```
asm ("or #0xFF, %0" : "+z"(x));
            /* x |= 0xff; No inputs or clobbers */
```

This ensures that `x` is loaded into R0 as required by the instruction. The `+` means that the register is both input (read) and output (written to).

Table 57. ST40 qualifiers (operand constraints)

Qualifier (Assembler register names)	Use (Corresponding ST40 register)
`"c"`	FPSCR
`"d"`	Double FP register
`"f"`	FP register
`"l"`	PR register
`"r"`	General purpose register
`"t"`	T bit - use to indicate side effecting T bit
`"w"`	FP0 (also known as FR0)
`"x"`	MAC register (MACH and MACL)
`"y"`	FPUL (FP communication register)
`"z"`	R0

The SuperH configuration defines special characters that are useful for ST40 code, these are listed in *Table 58* and examples are provided below.

Table 58. ST40 inline assembler template characters

Character	Description
`O`	Substitute a constant without the `#`. This is useful to emit a constant with `.long`. The value must be a constant or an expression that evaluates to a constant. The example below uses a function pointer.
`R`	Substitute the register name corresponding to the least significant 32 bits of a 64-bit value (irrespective of the endianness).
`S`	Substitute the register name corresponding to the most significant 32 bits of a 64-bit value (irrespective of the endianness).
`T`	Substitute the register name corresponding to the second 32-bit word of the 64-bit value (the same as `R` in big endian mode, and the same as `S` in little endian mode).

In the following examples, note how the template characters are placed between the `%` and the number. They may be used for parameters of any type, whether they be registers or memory addresses.

Example of %R and %S

This example demonstrates how to express the C expression `shreg = shreg << 1` in assembler, where `shreg` is of type `long long` (64-bit).

```
asm ("shll %R0; rotcl %S0" : "+r" (shreg) : : "t");
```

Example of %T

This example demonstrates the C expression `result = flag == 0` in assembler, where `flag` is of type `long long` (64-bit).

```
asm ("mov %1,%0; or %T1,%0; tst %0,%0 ; movt %0"
     : "=r" (result) : "r" (flag) : "t");
```

Note: *`%1` and `%T1` both refer to the same 64-bit variable, `flag`, each corresponding to a different 32-bit word.*

Example of %O

This example demonstrates a method of implementing the C expression `for (;;) fun ()` in assembler.

```
asm __volatile__ ("mov.l 0f, r1\n\t"
                  ".balign 4\n\t"
                  "mova 0f, r0\n\t"
                  "add r1, r0\n\t"
                  "braf r1\n\t"
                  "lds r0, pr\n"
                  "0: .long %O0 - 0b"
                  : : "i" (fun));
```

B.1.4 Compiler pragmas and attributes

The compiler supports the following pragmas and attributes. Note some pragmas can also be written as attributes.

Note: *These pragmas and attributes should not be used in OS21 applications as OS21 manages interrupts and traps.*

Pragma interrupt

This pragma specifies that a function is an interrupt handler, for example:

```
#pragma interrupt(fred)
int fred(int i);
```

The compiler alters the generated code as follows:

- uses RTE as the function return instruction (instead of RTS)
- executes an extended context save and restore to save the registers normally considered as scratch registers

Note: *This pragma must be specified before any of the following interrupt handler related attributes are used.*

interrupt_handler attribute

This is equivalent to the `interrupt` pragma and is specified as follows:

```
int fred(int i) __attribute__((interrupt_handler));
```

sp_switch attribute

This is used in conjunction with the `interrupt_handler` attribute to specify that the handler should be executed on an alternative stack. The compiler generates code to switch to and from this stack on function entry and exit respectively. The stack is named as a parameter to the `sp_switch` attribute as follows:

```
extern void *VBR_STACK;
int fred(int i) __attribute__((interrupt_handler,
                            sp_switch("VBR_STACK")));
```

This specifies that `VBR_STACK` is to be used as the interrupt stack.

naked attribute

Use this attribute to disable the generation of prologue and epilogue code to save and restore the *callee save* registers of a function, as defined by the ABI. See *Using the GNU Compiler Collection* for more information.

trap_exit attribute

The compiler generates a TRAPA instruction to exit the function rather than a standard RTE exit (the `trap_exit` attribute only applies to interrupt handlers). It also saves all registers before using them, even registers defined to be caller save.

```
#pragma interrupt(fred)
int fred(int i) __attribute__((trap_exit(42)));
```

The number is the parameter to the TRAPA instruction (the trap number).

Pragma trapa

This is equivalent to the `interrupt` pragma, except that it does not save extra registers.

```
#pragma trapa(func)
int func(int i);
```

Pragma GCC optimize

Use this pragma to set global optimization options for all functions defined in the source file from the point where this pragma is specified to the end of the file.

```
#pragma GCC optimize (string ...)
```

where *`string`* is a string of optimization options to apply to the functions that follow the pragma. More than one string can be specified. This is equivalent to adding the `optimize("`*`string`*`")` attribute to each function.

This pragma enables frequently executed functions to be compiled with more aggressive optimization to produce faster (but larger) code, whilst all other functions in the source file are called with less aggressive optimization to ensure that the code is kept as small as possible.

B.1.5 Link time optimization

The Link Time Optimizer (LTO) enables GCC to widen the scope of optimizations across multiple object files. For example, it allows a function to be inlined anywhere in an executable or relocatable library.

This section provides a high level view of the LTO in order to understand its limitations. For more information, see the relevant section of the *Using the GNU Compiler Collection* manual.

The LTO operates in two stages. Both stages are invoked using the `-flto` command line option.

1. The first stage is when compiling a source file. The resulting object file contains both the executable code plus the GIMPLE internal representation in dedicated `.lto` sections. The `.lto` sections increase the file size, but are not loaded into memory. This means that the object file can be used in exactly the same way as any other object file, including as inputs to relocatable and static libraries. The `.lto` sections are ignored unless the `-flto` option is also specified when linking.
2. The LTO itself is invoked by the linker by passing the `-flto` option on the link command line. It is possible to mix object files compiled without the `-flto` option (such as system libraries) with objects that were compiled with the `-flto` option. The compiler driver organizes the linker and the compiler invocations to construct the function dependencies, to compile and optimize the code and to perform the final link. The only difference is that the link takes longer than usual and uses more host memory.

Dead function removal

Function references are seen either from a non `-flto` compiled object file or from the GIMPLE internal representation. Consequently, functions that are not referenced are not emitted. This provides the same advantage as the linker garbage collector option (`--gc-sections`) but without the need to compile the functions into different sections.

The original method to perform dead function removal is:

```
sh4gcc -c -ffunction-sections a1.c a2.c
sh4gcc -mboard=board -Wl,--gc-sections a1.o a2.o
```

Note: This method is still supported by GCC, although the recommended method is to use LTO.

Dead function removal can now be performed automatically as follows:

```
sh4gcc -c -flto a1.c a2.c
sh4gcc -mboard=board -flto a1.o a2.o
```

The linker traverses the call graph starting from all entry points visible to the outside. This includes:

- the ELF entry point given with the `-e` linker option (the default is the symbol `start`) or the `ENTRY` directive in the linker script
- functions that are explicitly marked with `__attribute__((used))`

This may exclude many functions that are needed by the user but do not appear in the call graph. For example, a dump function that is only required for debugging or functions called from inline assembly code are not emitted. To force these functions to be emitted, they should be explicitly marked as such with `__attribute__((used))`.

In the following example, the function `func` would not be emitted without `__attribute__((used))` being specified:

```
int main(void)
{
  asm("mov.l   1f,r0");
  asm("jmp     @r0");
  asm("nop");
  asm(".align 2");
  asm("1: .long _func");
}

void __attribute__((used)) func(void)
{
...
}
```

Static functions

At the file level, static functions in C are visible inside their compilation unit. Because the LTO merges multiple compilation units, static functions are renamed to avoid name clashes and multiple definitions. The mangled name is specific to the implementation only and cannot be used to reference the function. This means that static functions cannot be referenced from inline assembly code if they have been compiled with `-flto`. They must be declared as global.

Debugging

When an application is linked with `-flto`, only limited debug information is generated. This means that only limited debugging is possible when the LTO is used.

The LTO and optimization levels

For the LTO to operate effectively, it is necessary to specify the same optimization options when linking as well as compiling. This is because the optimizations are performed twice, once to compile the object files and again to link the executable (by a call back to the compiler). For example, consider the following sequence:

```
sh4gcc -O2 -flto -c a1.c a2.c
sh4gcc -mboard=board -flto a1.o a2.o -o a.out
```

The executable `a.out` is not optimized because the `-O2` option is not specified. In order to ensure that the executable is optimized, include the `-O2` option in the link command line, as shown below:

```
sh4gcc -mboard=board -O2 -flto a1.o a2.o -o a.out
```

This requirement applies to all code generation options that affect optimization; that is, the `-O`, `-m` and `-f` options.

Note: *The optimization option specified when linking object files supersedes the optimization options used to compile the object files. For example, in the following sequence:*

```
sh4gcc -O3 -flto -c a1.c a2.c
sh4gcc -mboard=board -Os -flto a1.o a2.o -o a.out
```

the GIMPLE internal representation from files `a1.o` and `a2.o` are recompiled at optimization level `-Os` rather than `-O3` as originally compiled.

The following pitfalls and limitations must be taken into account.

- If no optimization options are specified when linking, the LTO does not optimize the resulting executable, even if the object files are compiled with optimization. To avoid this happening, specify the optimization options when linking.
- Mixing incompatible optimization flags between compiling object files and final link may result in unpredictable behavior. Examples of incompatible flags are those defining the ABI (for example, `-m4-single`) or if they have semantic impacts (such as `-fno-strict-aliasing` or `-fno-zero-initialized-in-bss`).
  - For example, the OS21 libraries are compiled with the option `-fno-zero-initialized-in-bss` to ensure that OS21 thread awareness is available immediately after loading the application code. This means that if the user wishes to debug an OS21 application (even at the basic level supported by the LTO), that application must either have the option `-fno-zero-initialized-in-bss` applied on the final link, or they must disable RTOS awareness (using the GDB command `disable rtos`) until `main` is called.
  - The restriction described above also applies when using OS21 Trace and OS21 Profiler with GDB.

Note: Adding the `-fno-zero-initialized-in-bss` option has the undesirable effect of moving all data objects that have been zero initialized from the BSS to the data section, which increases image size.

Builtin functions

Bultin functions are functions that are known internally by GCC. Such functions can be generated inline very late in the compilation flow, which means that redefining them in object files compiled with `-flto` conflicts with the inlined definition. For this reason, builtin functions do not appear in the list of functions referenced by the LTO (as this would mean they would be eliminated) but are still referenced nevertheless.

The functions that cannot be compiled with `-flto` include all of the functions defined as reserved in the *ISO/IEC standard 9899:1999 Programing languages -- C*. In addition, any functions that these reserved functions are dependent upon (directly or indirectly) cannot be compiled with `-flto`.

For this reason, the `-flto` option is limited to user code that is not used to compile runtime or system libraries.

Symbol redefinition

It is not possible to redefine a symbol after the LTO compilation stage. For example, the two methods traditionally used for redefining a symbol do not work for the following reasons.

- The linker `--wrap=`*`symbol`* option resolves any undefined symbol referenced in the module by `__wrap_`*`symbol`*. As all modules are merged into one by the LTO, the symbol is already defined and cannot be resolved with `__wrap_`*`symbol`*.
- The **sh4objcopy** `-redefine-sym` option allows the user to rename the symbols in the symbol table. This creates a conflict with the symbol being referenced from the GIMPLE internal object representation.

B.1.6 Stack overflow checking

The ST40 Micro Toolset supports the GCC `-fstack-protector` option to check for stack buffer overflows. The compiler achieves this by adding guard variables on the stack of functions with variables that are at risk of buffer overflows (such as functions that call `alloca` or have buffers larger than eight bytes). The guard variables are initialized with a *canary value* (that is, an arbitrary value that is known before the function is called and can be checked when the function returns). If the value of the guard variable is different when the function returns, this indicates that a buffer overflow has occurred and the stack protection support library raises an error. The error handling (and the initialization of the *canary value*) is handled by the **libssp** stack protection support library.

The version of the **libspp** library distributed with the ST40 Micro Toolset contains some modifications to improve its suitability for use in embedded systems. The main changes are the addition of overrides for initialization and error handling, as described in the following sections.

__guard_setup_override Initialize the canary value

Definition: `void __guard_setup_override(void **guard)`

Arguments:

`guard`	A pointer to a variable that contains the canary value.

Returns: None.

Description: This function, if defined, is called by `__guard_setup` in the **libssp** library. Define this function to set the canary value required for the guard variable. It can be a random or a fixed value.

The default is to use a fixed value of `0xFF0A0000`

__chk_fail_override User defined error handler

Definition: `void __chk_fail_override(void)`

Arguments: None

Returns: None

Description: This function, if defined, is called by `__chk_fail` in the **libssp** library. Define this function to handle a stack overflow error.

If the function is not defined, or if it returns, `__chk_fail` calls the GCC builtin function `__builtin_trap`. This executes a TRAPA instruction with an operand value of 42.

__stack_chk_fail_override User defined error handler

Definition: `void __stack_chk_fail_override(void)`

Arguments: None

Returns: None

Description: This function, if defined, is called by `__stack_chk_fail` in the **libssp** library. Define this function to handle a stack overflow error.

If the function if not defined, or if it returns, `__stack_chk_fail` calls the GCC builtin function `__builtin_trap`. This executes a TRAPA instruction with an operand value of 42.

B.1.7 Assembler specifics

The SH-4 assembler recognizes the command line options listed in *Table 59* in addition to the standard assembler options.

Table 59 lists the most useful ST40 `--isa` options. These options allow the assembler to do SH-4 specific optimizations (in conjunction with `--relax`). The assembler might determine this for itself, but only if there is an SH-4 specific instruction in the code. The `--isa` option allows any supported architecture variant to be selected.

Table 59. Assembler command line options

Option	Description
`--isa=sh4`	Assemble for a generic SH-4 core.
`--isa=st40-300`	Assemble for an ST40-300 series core.
`--isa=sh4-nofpu`	Assemble for a generic SH-4 core without an FPU. This is useful for the ST40-500 series cores. Any FPU instructions are rejected with an error.
`--isa=st40-300-nofpu`	Assemble for an ST40-300 series core without an FPU. Any FPU instructions are rejected with an error.
`--isa=sh4-nommu-nofpu`	As for `--isa=sh4-nofpu` but MMU instructions are also rejected. This is useful for the ST40-400 series cores.

When the `--isa` option is not specified, the assembler selects the most appropriate architecture for an object file based on the instructions used. Therefore, if no SH-4 specific instructions are used, it is quite normal for an SH-4 object file to be set as `sh4-nofpu`, `sh3` or an even earlier variant.

Although the assembler may set the architecture for a non SH-4 architecture, it should be noted that the compiled code still implements the ABI of the originally specified architecture. In general, it is not the case that two object files purporting to be for the same architecture are compatible. For example, an object file intended for use with the `sh4-nofpu` architecture uses a different ABI to an object file intended for use with an SH-4 with an FPU, but does not happen to use any floating-point instructions (and is therefore also set to the `sh4-nofpu` architecture).

Note: The linker almost always links such object files without error, and in some limited cases, the resulting application may even execute successfully.

The SH-4 assembler recognizes the pseudo-opcodes listed in *Table 60*.

Table 60. Specific pseudo-operations

Pseudo opcodes	Action

`.long` *`value`*
`.int` *`value`*
	Allocate 4 bytes

`.word` *`value`*
`.short` *`value`*
	Allocate 2 bytes

`.big`	Specify big endian
`.little`	Specify little endian
`.heading "`*`name`*`"`	Specify *`name`* as name of listing file

Table 60. Specific pseudo-operations (continued)

Pseudo opcodes	Action
`.page`	New page in listing file
`.uses`	Used to label a call instruction for linker relaxation. (Used by compiler to support `-mrelax`)

`.uaword` *`value`*
`.2byte` *`value`*
	Unaligned 2-byte allocation

`.ualong` *`value`*
`.4byte` *`value`*
	Unaligned 4-byte allocation

`.uaquad` *`value`*
`.8byte` *`value`*
	Unaligned 8-byte allocation

The assembler supports the assembler syntax as defined in the *ST40 core and instruction set architecture reference manual* (7182230), in addition lower case instructions and register names are supported. It should be noted that for the SH-4 family of cores, numeric literals must be prefixed by a `#`.

Table 61 lists the register names recognized by the assembler.

Table 61. Recognized register names

Register name	Use
R0 to R15	General purpose registers
FR0 to FR15	Floating-point registers
DR0, DR2, DR4, DR6, DR8, DR10, DR12, DR14	Double-precision floating-point registers
SR, GBR, SSR, SPC, SGR, DBR	Control registers
R0_BANK to R7_BANK	Register in other (non selected) bank
XF0 to XF15	Floating-point extended registers
FV0, FV4, FV8, FV12	Floating-point register vectors, 4-way
XD0, XD2, XD4, XD6, XD8, XD10, XD12, XD14	Double-precision extended registers
XMTRX	4 x 4 column, single-precision matrix
MACH, MACL, PC, FPUL, PR, FPSCR	System registers

The assembler supports the following pseudo-instructions.

`mov.l` *`symbol`*`,` *`rn`*

If *`symbol`* is a label that is reachable from this instruction, the instruction is expanded to a PC-relative `mov.l` instruction. That is, an instruction in the format:

```
mov.l @(disp,pc), rn
```

The *`symbol`* must be 4-byte aligned as this is a requirement for the encoding of this instruction.

`mov.w` *`symbol`*`,` *`rn`*

If `symbol` is a label that is reachable from this instruction, it is expanded to a PC-relative `mov.w` instruction. The *`symbol`* must be 2-byte aligned.

B.1.8 Linker relaxation

This is a process carried out by the linker to shorten branches between code in different compilation units. Linker relaxation applies to conditional (with and without delay slots) and unconditional jumps. This option is used to reduce code size and to improve code performance.

In order for this optimization to be effective, it is important that the relaxation option `-mrelax` is applied to all compilations in addition to the final link.

B.1.9 Floating-point behavior

When executing floating-point instructions, the precision of the operation (either single-precision or double-precision) is controlled by setting the PR bit in the floating-point status/control register, FPSCR. This is in contrast to other architectures where the precision is indicated by the floating-point instructions that are used.

The default C runtime bootstrap sets the initial value of the precision to double. This is known as the **pervading precision**. When single-precision operations are required, the compiler must generate code to switch to single-precision. This behavior is inefficient when the majority of floating-point operations are single-precision, therefore the default behavior can be overridden using one of the following options:

```
-m4-single
-m4-100-single
-m4-200-single
-m4-300-single
```

When these options are used, the pervading precision is set to single and the compiler generates code to switch it to double when necessary.

It is also possible to ignore double-precision altogether using one of the following options:

```
-m4-single-only
-m4-100-single-only
-m4-200-single-only
-m4-300-single-only
```

These options cause the compiler to convert all variables of `double` type to `float` type. Obviously this can have a serious effect on the accuracy of the calculations, but can also improve the performance of the program.

Note: *All of these options change the ABI and therefore all object files and libraries in an application must be compiled with the same option. Additionally, the same options must be supplied to GCC at link time as well as at compile time to ensure that the correct run-time libraries are selected.*

The default C runtime bootstrap does not set the ENABLE bit in the FPSCR register. Therefore, by default, no floating-point unit (FPU) exceptions are generated. Instead, the FPU selects a suitable value, such as INF (infinity) or NaN (not a number). The FPSCR register may still be queried as described in the architecture manual.

B.1.10 Speed and space optimization options

To obtain the best performing code, use the following compiler and linker options:

`-O3` All compiler global and local optimizations.

`-funroll-loops`
`-funroll-all-loops` (Some experimentation may be required to find which of the two options is preferable). These options unroll loops and provide longer straight-line code sequences that are more suitable for compiler optimization.

`-mrelax` Used to shorten branches. To be effective, the option must be used consistently for each of the compile, assemble and link phases.

For minimal code size, use the following compiler and linker options:

`-Os` This instructs the compiler to optimize for space.

`-malign-small-blocks=`*n* Aligns labels on a cache line if the block is larger than *n* instructions. The default is 16, so *n* = 32 will reduce code size. *n* = 0 aligns every branch target, and will increase code size.

`-mrelax` Used to shorten branches. To be effective, the option must be used consistently for each of the compile, assemble and link phases.

B.2 Cross development tools reference

B.2.1 Command scripts

The GDB command scripts provide the required configuration information so that GDB can connect to a target (either silicon or simulator).

The command scripts are located in the `sh-superh-elf/stdcmd` subdirectory of the release installation directory. For more information on these files, see *ST40 Micro Toolset GDB command scripts* (8045872).

B.2.2 ST TargetPack

The ST TargetPack is an alternative method for describing target systems using an ST SoC. The ST TargetPack for a specific target provides all the information that a host program (such as a debugger) needs to be able to connect to that target.

For more information on ST TargetPacks, see *ST TargetPack user manual* (8020851).

B.2.3 Memory mapped registers

The user commands for connecting to a target also define, as GDB convenience variables, symbolic names for the memory mapped registers of the target. This includes:

- all of the architecturally defined ST40 peripherals
- all the peripherals configured during connection
- a selection of other common peripherals (for example, LMI, EMI and PCI)

These symbolic names may be used interactively or by GDB user defined commands to read from and write to the memory mapped registers instead of explicitly specifying their address.

The symbolic names for the memory mapped registers use a standard naming convention, where the register name is composed of the peripheral group name followed by the register name (separated by an underscore). For example, the ST40 cache control register CCR, which is a member of the CCN peripheral group, is defined as `$CCN_CCR`. The register name is prefixed with `$` to indicate that the symbol is a GDB convenience variable and not a symbol in the target application.

Note: 1 *The list of register names can be found in the* `register40.cmd` *GDB command script (see Section B.2.1: Command scripts on page 262).*

2 *ST TargetPacks also contain definitions of the SoC registers. See Appendix F: ST TargetPack plugin on page 282.*

3 *The GDB* `show convenience` *command can be used to display the currently defined convenience variables.*

Reading a memory mapped register

The following example illustrates using the GDB `x` command to read the ST40 cache control register via the pre-defined GDB convenience variable.

```
(gdb) x/wx $CCN_CCR
```

Writing to a memory mapped register

The following example illustrates using the GDB `set` command to change the ST40 cache control register via the pre-defined GDB convenience variable:

```
(gdb) set *$CCN_CCR = 0x00000909
```

The pointer dereference (`*`) operator is required since the GDB `set` command has to dereference the pointer `$CCN_CCR` and assign a value to it.

Note: The GDB variable containing the address of a memory mapped register is not read-only and will be changed if the `*` is omitted.

B.2.4 Silicon specific commands

GDB hardware breakpoints and watchpoints are only supported on silicon and not in a simulation environment. However, they are limited by the debug hardware available within the ST40.

Hardware breakpoint support

A hardware breakpoint is set using the `hbreak` command. The `hbreak` command is similar to the `break` command (that sets software breakpoints):

```
hbreak function|line|file:line|*address
```

The ST40 can only support up to three hardware breakpoints at the same time. If used in conjunction with hardware watchpoints it is only possible to have a combined total of three hardware breakpoints and watchpoints at any one time.

Note: The number of software breakpoints is not limited by the hardware.

Hardware breakpoints should be used when debugging applications booting from ROM as software breakpoints are not usable when the ST40 executes code from ROM.

Hardware watchpoint support

To set a hardware watchpoint, use one of the commands listed in *Table 62*. Where *`location`* is a location expression (for example, an address or a symbolic object name).

Table 62. Hardware watchpoint commands

Command	When triggered
`watch` *`location`*	Write accesses only.
`rwatch` *`location`*	Read accesses only.
`awatch` *`location`*	Both read and write accesses.

The ST40 can only support up to three hardware watchpoints at the same time. If used in conjunction with hardware breakpoints it is only possible for a combined total of three hardware breakpoints and watchpoints to be set at any one time.

Additionally, the ST40 hardware watchpoints have the capability to refine a watchpoint to only report matches when the core accesses the watched address range using a specific access size and ignoring accesses of other sizes. The core access sizes supported by the ST40 hardware watchpoints are 1, 2, 4 and 8 bytes. This feature is controlled using the `use-watchpoint-access-size` command. *Table 63* lists the access size settings of the `use-watchpoint-access-size` command.

Table 63. use-watchpoint-access-size access size modes

Mode	Description
`off`	Access size checking is disabled. Any core access matching the watch conditions will be reported.
`on`	Access size is enabled and is derived from the watched region size(1). This is the default.
`1`	Enable 8-bit access size checking (for example, `MOV.B Rm, @Rn`).
`2`	Enable 16-bit access size checking (for example, `MOV.W Rm, @Rn`).
`4`	Enable 32-bit access size checking (for example, `MOV.L Rm, @Rn`).
`8`	Enable 64-bit access size checking (for example, `FMOV` when FPSCR.SZ=1)(2).

1. In this mode, access size checking is only performed if the size of the watched region is 1, 2, or 4 bytes; if not then access size checking is disabled for the watched region. Checking for 64-bit access sizes for a watch region of 8 bytes (or any watch region size) is only supported by using the `use-watchpoint-access-size` command to set the access size checking mode to 8.
2. See the *ST40 Core Architecture Manual* for further details.

The access size checking mode set by the `use-watchpoint-access-size` command is global and is only applied to the hardware watchpoints when a target is restarted. Therefore, the order in which the `use-watchpoint-access-size` and hardware watchpoint commands (see *Table 62*) are used is unimportant; the access size checking mode for hardware watchpoints only takes effect when the target is restarted[a].

Note that the ST40 hardware watchpoints have limited capability in the regions they are able to watch. If a watch region greater than 4 bytes is requested then the ST40 hardware is only able to watch a fixed range of memory region sizes and alignments. This is because the address comparator of the ST40 watchpoint hardware only supports the following options:

- all bits compared
- upper 22 bits compared (1 Kbyte page)
- upper 20 bits compared (4 Kbyte page)
- upper 16 bits compared (64 Kbyte page)
- upper 12 bits compared (1 Mbyte page)
- no bits compared (any address matches)

As a result, if the requested watch region does not match one of the above page sizes and alignments the GDB implementation selects the page size and alignment that covers the address range of the watch region. As a consequence, this results in spurious watchpoints being reported for addresses outside the requested watch region (the worst case being for watch regions that do not fit within a 1 Mbyte page causing watchpoints to be reported for every memory access). Note that for watchpoints set using the `watch` command GDB only reports a watchpoint if the value of the data in watch region has been changed and not just written to.

GDB also supports software watchpoints. Using software watchpoints has a significant impact upon the performance of the target.

[a] The reason for this is because GDB automatically clears breakpoints and watchpoints when a target stops and resets them only when a target is restarted (except when the `set breakpoint always-inserted` option is set to `on`). Therefore the `use-watchpoint-access-size` mode only takes effect when the target is restarted.

B.3 Embedded features

B.3.1 Default C run-time bootstrap

The default C run-time bootstrap carries out a number of actions to set-up the ST40 for program execution and to shut-down the program on termination. The steps below are carried out by the default C run-time bootstrap.

1. Set up stack pointer (R15) from the value of the symbol `_stack`.
2. Zero the `bss`. (Global data is initialized to zero.)
3. Set FPSCR, the floating-point control register, according to the pervading precision.
4. Set up the exception handlers by setting VBR and by setting the status register (SR) to allow exception handlers to be called. By default, the power on value of the status register blocks exceptions causing them to raise a manual reset. Trap handlers are set for general exceptions.

Note: For OS21 applications, once OS21 has started, all exceptions are handled using the OS21 exception handler and these embedded features no longer apply (see also Section B.3.2).

5. Call `main`.

Note: The C run-time bootstrap does not set up any virtual address mapping.

On return from `main()`, the `exit()` function is called with the return code from `main()`.

B.3.2 Trap handling

When a general exception occurs and when it is vectored through the default exception handlers, the C run-time bootstrap returns the exception code as the exit code of the program.

It is possible to use the debugger to catch the exception by setting a breakpoint on the function `_superh_trap_handler`. The debugger is then able to provide a stack trace to the location of the exception.

The trap handler function has one parameter that can be examined in the debugger. This is the value of the EXPEVT register at the point of the exception and provides the reason for the exception.

__builtin_trap()

The implementation of the GCC built-in function `__builtin_trap()` has been changed to issue a TRAPA instruction with an operand value of 42 instead of a call to the `abort()` function. This has been done to make the function more suitable for OS21 and bare machine applications, where the exception can be caught and handled by the user.

Appendix C Performance counters

Performance counters are an ST40 hardware feature to aid the debugging and analysis of an application, by providing the ability to count execution cycles or the occurrences of several different kinds of events during the execution of an application.

The ST40 provides a pair of 48-bit counters, designated as `counter1` and `counter2`. These counters can be configured to count the occurrence of a variety of useful events, or to count cycles while the ST40 core is in certain states.

The counters can be either started and stopped manually from GDB or automatically by specifying start and stop triggering addresses, which when the instructions at these addresses are executed, start and stop the counters.

Each counter can be individually configured to start and stop automatically at these triggering addresses. However, the ST40 only supports a single pair of start and stop trigger addresses which are shared between both performance counters. As a consequence, both counters start and stop at the same triggering addresses.

The counters can be individually configured to count any one of the 34 different events or states listed in *Section C.1*, where the type of event or state is identified by either its numeric code or its corresponding symbolic name (which is a mnemonic for the event).

The counter modes (see *Section C.1*) in which the counter counts in cycles instead of discrete events, support the following two forms of cycle counting.

CPU The counter counts the number of CPU clock cycles (`ICLK`) where the count is incremented by 1 for every CPU clock cycle while the relevant condition applies.

BUS The counter counts the number of BUS clock cycles. This is defined as the inverse ratio of the CPU clock (`ICLK`) to the bus clock (`BCLK`) multiplied by 24 (`BCLK / ICLK * 24`). The BUS clock increment is therefore dependent on the `ICLK` to `BCLK` ratio and will be 12 for a 2:1 ratio, 8 for a 3:1 ratio, 6 for a 4:1 ratio, 4 for a 6:1 ratio and 3 for an 8:1 ratio. These increments approximate to real time `T` where:
`T = BCLK / 24 * count`.

Note: The performance counters do not have any impact on CPU performance.

The performance counter features are accessible though the GDB `perfcount` command.

C.1 Performance counter modes

The performance counter modes and their symbolic names (which are case sensitive) are listed in *Table 64*.

Table 64. Performance counter modes

Code	Symbol	Countable event	Count or cycle	Notes
0	`nop`	(nop)	N/A	
1	`oar`	Operand Access (Read with cache)	Count	
2	`oaw`	Operand Access (Write with cache)	Count	
3	`utlb`	UTLB miss	Count	
4	`ocrm`	Operand Cache Read Miss	Count	
5	`ocwm`	Operand Cache Write Miss	Count	
6	`if`	Instruction Fetch (with cache)	Count	Instructions fetched in pairs
7	`itlb`	Instruction TLB miss	Count	
8	`icm`	Instruction Cache miss	Count	
9	`aoa`	All Operand Access	Count	
10	`aif`	All Instruction Fetch	Count	Instructions fetched in pairs
11	`ram`	On-chip RAM operand access	Count	
12	`io`	On-chip IO space access	Count	
13	`oarw`	Operand Access (Read & Write with cache)	Count	Equivalent to `oar + oaw`
14	`ocrwm`	Operand Cache (Read & Write) Miss	Count	Equivalent to `ocrm + ocwm`
15	`bi`	Branch instruction issued	Count	
16	`bt`	Branch taken	Count	
17	`bsr`	BSR/BSRF/JSR instruction issued	Count	
18	`ii`	Instruction issued	Count	
19	`2ii`	Two instructions issued simultaneously	Count	
20	`fpu`	FPU instruction issued	Count	
21	`int`	Interrupt (normal)	Count	
22	`nmi`	Interrupt (NMI)	Count	
23	`trapa`	TRAPA instruction executed	Count	
24	`ubca`	UBC-A channel match	Count	
25	`ubcb`	UBC-B channel match	Count	
26	`icf`	Instruction Cache Fill	Cycle	

Table 64. Performance counter modes (continued)

Code	Symbol	Countable event	Count or cycle	Notes
27	ocf	Operand Cache Fill	Cycle	
28	time	Elapsed TIME	Cycle	
29	pfi	Pipeline Freeze due to cache miss Instruction	Cycle	
30	pfo	Pipeline Freeze due to cache miss Operand	Cycle	
31	pfb	Pipeline Freeze due to Branch instruction	Cycle	
32	pfr	Pipeline Freeze due to CPU register	Cycle	
33	pff	Pipeline Freeze due to FPU resource	Cycle	

The `oar`, `oaw`, `oarw` and `if` modes are only applicable for accesses and fetches in cacheable areas of the address map while caches are enabled. They are undefined when a PMB or TLB controls the caches. (This is not the case for the ST40-300 series cores.)

The `icm` mode includes fetches from noncacheable memory (effectively this counts instruction fetches taking > 1 cycle).

The `bi` and `bt` modes count all branch instructions (BF, BF/S, BT, BT/S, BRA, BRAF and JMP), except for the special case of a BF or BT instruction with a displacement of zero.

For the `ii` and `fpu` modes, the count is incremented by 2 for a pair of (floating point) instructions simultaneously dual-issued.

The `pfb` mode counts all branch instructions regardless of displacement. The count is 1 cycle per branch except when a delay slot instruction avoids the pipeline stall. If the target instruction is not in the cache, stalls due to instruction cache refill cycles are counted with `pfi`.

The `pfi` and `pfo` modes also count the wait time for on-chip RAM and I/O space accesses. The counts also include freeze cycles for accesses and fetches performed without caches.

The `pff` mode only counts when no instructions are issued due to FPU resource contention. If one instruction can be issued in a given cycle the count is not incremented.

The counts in all modes must be considered approximate as they may contain errors. For example, the presence of exceptions causes overestimation in the count. In addition there may be a mis-counting of events at the start and the end of the performance measurement. For these reasons, the counts can be unreliable if performed over a short period or application range.

Note: The `oar`, `oaw`, `if` and `oarw` counter modes include all accesses, including cache misses (see Table 64).

C.2 The perfcount command

The `perfcount` command is enabled automatically when the host connects to a target. It can also be enabled by issuing the GDB command `enable_performance_counters` (as defined in the `perfcount.cmd` GDB command script).

The `perfcount` command has the following format:

```
perfcount subcommand options
```

This command controls the performance counter function specified by *subcommand* and *options*.

The subcommands supported by the `perfcount` command are listed in *Table 65*.

Table 65. perfcount subcommands

Subcommand	Options	Description
`help`	*[subcommand]*	Display help for the `perfcount` command. If a *subcommand* is specified then more detailed help for *subcommand* is displayed.
`status`	`counter1|counter2|all`	Display the configuration of the specified counter or all counters.
`enable`	`counter1|counter2|all`	Enable counting for the specified counter or all counters.
`disable`	`counter1|counter2|all`	Disable counting for the specified counter or all counters.
`trigger`[1]	`counter1|counter2|all |none` *start-addr stop-addr*	Set the start and stop trigger addresses of the specified counter or all counters to *start-addr* and *stop-addr*. Alternatively, if `none` is specified, remove any previously specified addresses. When `none` is specified, the counter is always on if enabled or off if disabled.
`mode`	`counter1|counter2|all` *mode*	Set the mode of the specified counter or all counters to *mode*, where *mode* is one of the numeric codes or its corresponding symbolic name listed in *Table 64 on page 267*. If *mode* is not recognized then `nop` is assumed.
`display`	`counter1|counter2|all`	Display the value of the specified counter or all counters.
`reset`	`counter1|counter2|all`	Reset the value of the specified counter or all counters to zero.
`return`	`counter1|counter2` *variable*	Return the value of the specified counter to a GDB convenience variable or target variable named *variable*.
`clock`	`counter1|counter2|all cpu|bus`	Set the cycle count mode of the specified counter or all counters to `cpu` or `bus`.

1. Do not use this subcommand in conjunction with hardware breakpoints and watchpoints as they use the same resources and this will cause conflicts.

Appendix D Profiler plugin

Profiling is a performance analysis technique that identifies the areas in an application where the CPU spends most time. Having identified these areas, it is then possible to target optimization efforts on the specific parts of the code that will yield the greatest benefit in terms of improving performance.

When a connection is made to the target using an ST Micro Connect, commands may be issued through GDB to instruct the ST Micro Connect to collect sampling information about a running application. This data is stored in a file and can then be analyzed using a profiling tool (such as **STWorkbench** or **sh4gprof**).

The profiler plugin provides two different types of profiling.

trace The profiler samples the PC over a given period, time stamping each sample. This method provides a view of the application’s activities over a period of time. See *Section D.2: Trace profiler output format on page 273*.

range The profiler accumulates the number of times that a particular region of the application’s code is executed (in the manner of **gprof**; see the *GNU gprof* documentation for more details). See *Section D.3: Range profiler output format on page 274*.

Profiling operates in one of three modes.

none In this mode, the profiler collects samples only when the target stops at a breakpoint or an I/O request.

udi This mode provides non-intrusive sampling to obtain the current address on the instruction bus.

interrupt This mode stops the target to read the PC directly before continuing. This mode has a significant impact on the real-time performance of the target, but it has the advantage of being able to read the PC directly.

The ST Micro Connect profiler features are accessible through the GDB `profiler` command.

D.1 Profiler plugin reference

The `profiler` command is enabled automatically when the host connects to a target. It can also be enabled by issuing the command `enable_profiler` (defined in the `profiler.cmd` GDB command script).

The `profiler` command has the following format:

```
profiler subcommand [options]
```

This command controls the profiler function specified by *subcommand* and *options*.

The subcommands supported by the `profiler` command are listed in *Table 66*.

Table 66. Profiler subcommands

Subcommand	Option	Description
`help`	[*subcommand*]	Display help for the `profiler` command. If a *subcommand* is specified then more detailed help for the *subcommand* is displayed.
`enable`		Start the profiler on the STMC the next time the target is re-started. Samples are **only** taken and stored by the STMC while the target is running. When the target is stopped, no samples are taken.
`disable`		Stop the profiler on the STMC. Stopping the profiler implies a `reset`.
`reset`		Discard the stored profiler data on the STMC.
`display`		Display the profiler data stored on the STMC.
`save`|`append`	*file*	Save or append the profiler data stored on the STMC to *file*.
`gmonout`	*file*	Save the range profiler data stored on the STMC to *file* using the **gprof** compatible `gmon.out` file format.
`mode`	`none`|`udi`|`interrupt`	Set the profiler sampling mode: – `none` only records samples when the target stops at a breakpoint or an I/O request (this is the default) – `udi` records samples using the non-intrusive method of sampling the PC – `interrupt` records samples by briefly stopping the target to sample the PC
`period`	*delay*	Set the minimum sampling period for the profiler. The *delay* period can be specified in seconds (s), milliseconds (ms) or microseconds (us) by using the appropriate suffix. If no suffix is specified, microseconds are assumed. If period is not specified, profiling is effectively disabled. It is therefore mandatory to set the sampling period.

Table 66. Profiler subcommands

Subcommand	Option	Description
type	none|trace|range	Set the type of profiler to be used: – none indicates that no profiler is to be used (this is the default) – trace enables the trace profiler where each sample is time stamped (see *Section D.2: Trace profiler output format on page 273*) – range enables the sampling profiler which increments a counter for an address range each time a sample is taken (see *Section D.3: Range profiler output format on page 274*)
trace	*size*	Set the maximum number of samples to store on the STMC. If insufficient space is available on the STMC to store the specified number of samples, profiling is effectively disabled. When the sample buffer is full, the oldest samples are discarded; therefore only *size* most recent samples are returned. The trace subcommand is mandatory when the type of the profiler is set to trace.
range	*size* [*startaddr endaddr*]	Set the *size*, in number of instructions, of the slot in the application's address range to associate with a counter. If insufficient space is available on the STMC to store the counters required for the specified address range, profiling is effectively disabled. The default address range for an application is determined by the __stext and __etext symbols (placed by the linker), but this may be overridden by specifying the start and end addresses explicitly. The start and end addresses can be specified symbolically or with absolute addresses. The range subcommand is mandatory when the type of the profiler is set to range.

D.2 Trace profiler output format

If the profiler type is set to `trace` (using the command `profile type trace`), the profiler trace report consists of a header line followed by a list of timestamped, sampled PC addresses.

The header line has the following format:

Trace Profiler (saved = *saved-records*, total = *total-records*, time = *end-time*)

where:

- *saved-records* is the number of records saved in the buffer
- *total-records* is the number of records captured since this profiling session started
- *end-time* is the time of the the last record captured since this profiling session started (`profiler reset`)

Using the trace profiler between two breakpoints provides a simple way to obtain an approximation of the elapsed time between two points in an application. To do this, start (or reset) the profiler at the first breakpoint, then display the profiler data at the second breakpoint. The *end-time* value gives the time elapsed between the two breakpoints.

If *saved-records* is less than *total-records*, the sample buffer has wrapped. The number of discarded records is *total-records - saved-records*.

The remainder of the profiler output is a list of PC samples. The number of samples is equal to *saved-records*. The list has the following format:

accumulated-delta address function [at *location*]

where:

- *accumulated-delta* is the accumulation of the time delta between samples since profiling started
- *address* is the sampled PC address
- *function* is the name of the function at the given address (`??` indicates that the function name is unknown)
- *location* is the source location of the given address, if known

Figure 36 provides an example of the output displayed for the `profile type trace`.

Figure 36. Example profile type trace output

```
Trace Profiler (saved = 232, total = 232, time = 53393)
0000000000 0x84016f70, fn_0_993 () at fn2_0.c:6956
0000000171 0x84016e68, fn_0_990 () at fn2_0.c:6935

...

0000053206 0x84001bd4, fn_0_5 () at fn2_0.c:42
0000053393 0x84001a24, fn_0_0 () at fn2_0.c:5
```

D.3 Range profiler output format

If the profiler type is set to `range` (using the command `profile type range`), the profiler trace report consists of a header line followed by a list of sample counters, each representing a range of program memory. For each sample taken, the profiler increments the counter for the address range (slot) where the PC is currently located.

The header line has the following format:

`Range Profiler (range = `*`address..address`*`, step = `*`step`*`, slots = `*`slots`*`, rate = `*`µs`*`us per sample)`

where:

- *`address..address`* is the start and end address of the memory range
- *`step`* is the size of each slot in bytes (as set by the `range` subcommand)
- *`slots`* is the total number of slots
- *`µs`* is the sampling rate, in microseconds per sample

The remainder of the profiler output has the following form:

`count address``:`*`address`*`, `*`function`*

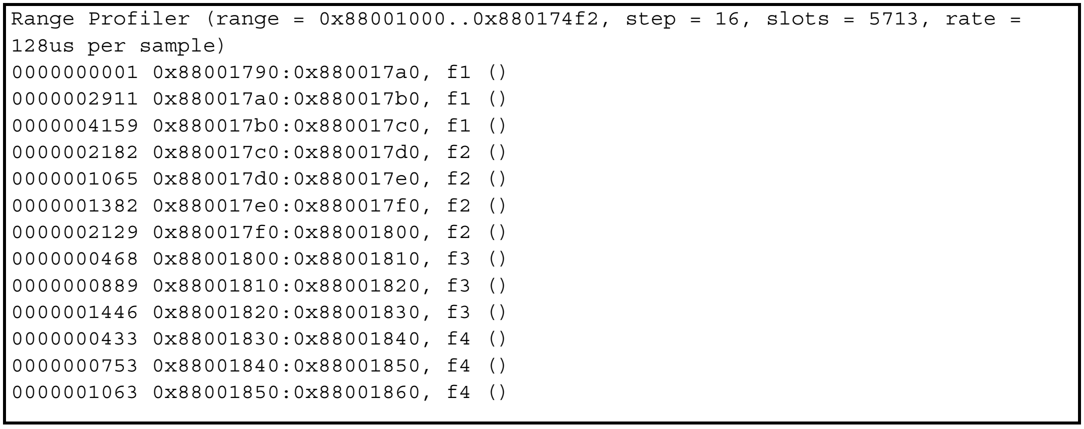
where:

- *`count`* is the sample count obtained for the given address range (slot)
- *`address`*`:`*`address`* is the start and end address of the slot
- *`function`* is the name of the function in which the slot is located (`??` indicates that the function name is unknown)

Note: The report displays only non-zero sample counters.

Figure 37 provides an example of the output displayed for the `profile type range`.

Figure 37. Example profile type range output



```
Range Profiler (range = 0x88001000..0x880174f2, step = 16, slots = 5713, rate =
128us per sample)
0000000001 0x88001790:0x880017a0, f1 ()
0000002911 0x880017a0:0x880017b0, f1 ()
0000004159 0x880017b0:0x880017c0, f1 ()
0000002182 0x880017c0:0x880017d0, f2 ()
0000001065 0x880017d0:0x880017e0, f2 ()
0000001382 0x880017e0:0x880017f0, f2 ()
0000002129 0x880017f0:0x88001800, f2 ()
0000000468 0x88001800:0x88001810, f3 ()
0000000889 0x88001810:0x88001820, f3 ()
0000001446 0x88001820:0x88001830, f3 ()
0000000433 0x88001830:0x88001840, f4 ()
0000000753 0x88001840:0x88001850, f4 ()
0000001063 0x88001850:0x88001860, f4 ()
```

D.4 ST Micro Connect configuration options

The profiling data is collected by the ST Micro Connect. The profiler can also be controlled by issuing ST Micro Connect configuration options.

Table 67 lists the ST Micro Connect configuration options and their equivalent profiler sub-commands, as listed in *Table 66 on page 271*.

Table 67. STMC configuration options

Configuration option	Equivalent to
`stmcconfigure profiler=on`	`profiler enable`
`stmcconfigure profiler=off`	`profiler disable`
`stmcconfigure profiler=reset`	`profiler reset`
`stmcconfigure profiler.mode=`*`mode`*	`profiler mode` *`mode`*
`stmcconfigure profiler.period=`*`delay`*	`profiler period` *`delay`*
`stmcconfigure profiler.type=`*`type`*	`profiler type` *`type`*
`stmcconfigure profiler.type.trace=`*`size`*	`profiler trace` *`size`*
`stmcconfigure profiler.type.range=`*`startaddr`*`:`*`endaddr`*`:`*`bytes`*[1]	`profiler range` *`size startaddr endaddr`*

[1] Unless overridden, *`startaddr`* is the address of the `__stext` symbol and *`endaddr`* is the address of the `__etext` symbol. *`bytes`* is the size, in bytes, of the number of instructions specified by *`size`*. `startaddr:endaddr:bytes` must have the form `0x`*`hexvalue`*`:0x`*`hexvalue`*`:0x`*`hexvalue`*.

D.5 Examples

The following GDB command script shows how to configure the ST Micro Connect to use the trace profiler in non-intrusive mode, automatically appending the results to a file every time the program stops and then resetting the profiler:

```
profiler mode udi
profiler period 1ms
profiler type trace
profiler trace 65536
profiler enable

define hook-stop
  profiler append "a.dat"
  profiler reset
end

continue
```

The following script describes a similar example that uses the range profiler, except that the results file and the `gmon.out` file are overwritten each time the target stops and the profiler data continues to accumulate on the ST Micro Connect:

```
profiler mode udi
profiler period 1ms
profiler type range
profiler range 8
profiler enable

define hook-stop
  profiler save "a.dat"
  profiler gmonout "gmon.out"
end

continue
profiler disable
```

To generate a report using the **sh4gprof** tool, use the following command line:

```
sh4gprof --no-graph a.out gmon.out
```

The `--no-graph` option is required because the `gmon.out` file produced by the range profiler does not contain call graph information.

Appendix E Branch trace buffer

The branch trace buffer is an ST40 hardware feature intended to aid debugging, showing the flow of control during execution of a program by recording the non-sequential updates of the program counter (PC).

The ST40 branch trace buffer is an eight level deep FIFO buffer, which stores the source and destination addresses for the last eight branches. The branch trace buffer can be configured for all branches, a class of branches or a combination of branch classes. The defined branch classes are general, subroutine and exception (see *Section E.1: Branch trace buffer modes* for further details).

Note: The collection of branch information does not have any impact on CPU performance. By default, the branch trace buffer is enabled and configured to trace all branches.

The branch trace buffer features are accessible though the GDB `branchtrace` command.

E.1 Branch trace buffer modes

The branch trace buffer can be configured to trace all branches or any combination of the traceable branch classes. The traceable branch classes and their symbolic names are listed in *Table 68*.

Table 68. Traceable branch classes

Traceable event	Symbol	Description
General branches	`gn`	BF[1], BT[1], BF/S, BT/S, BRAF and JMP.
Subroutine branches	`sb`	BSR, BSRF, JSR and RTS.
Exception branches	`ex`	All exceptions, interrupts and RTE.
All branches	`all`	All branch classes traced.
No branches	`none`	No branches traced.

1. Except for the special case where the displacement is zero; that is, the instructions `BF 0` or `BT 0` are not recorded in the buffer. This is a limitation of ST40 branch trace.

When using the `branchtrace` command, the branch trace classes `gn`, `sb` and `ex` can be combined with a `+` operator. For example, use `gn+ex` to trace general and exception branches; or `sb+gn` to trace subroutine branches and general branches.

Note: `all` is equivalent to `gn+sb+ex`.

E.2 The branchtrace command

The `branchtrace` command is enabled automatically when the host connects to a target. It can also be enabled by issuing the GDB command `enable_branch_trace` (defined in the `brtrace.cmd` GDB command script).

The `branchtrace` command has the following format:

`branchtrace` *`subcommand options`*

This command controls the branch trace buffer function specified by *`subcommand`* and *`options`*.

Note: For convenience, the `branchtrace` *command is aliased to the* `brt` *command.*

The subcommands supported by the `branchtrace` command are listed in *Table 69*.

Table 69. Branchtrace subcommands

Subcommand	Options	Description
`help`	*`[subcommand]`*	Display help for the `branchtrace` command. If a *`subcommand`* is specified then more detailed help for the *`subcommand`* is displayed.
`append`	*`file`* `[reset]`	Append the branch trace data to the end of *`file`*. The file is appended in the same format as the `display` command. If `reset` is specified, then append the initial branch trace contents instead of the current contents.
`appendsim`	*`file`* `[reset]`	Append the branch trace data to the end of *`file`*, in the same format as the simulator `sim_branchtrace` command (see *sim_branchtrace command on page 108*). If `reset` is specified, then append the initial branch trace contents instead of the current contents.
`decode`	`on	off`	Switch the decoding of ST40 opcodes on or off when reporting the branch type. The default is `on`.
`display`	`[reset]`	Display the branch trace buffer contents. If `reset` is specified, then display the initial branch trace contents instead of the current contents.
`mode`	*`mode`*	Set the mode of the branch trace buffer, where *`mode`* is one of the symbols listed in *Table 68* (or a combination of symbols concatenated with `+`).
`remote`	*`size`* `| on	off	stopon	stopoff`	Configure the continuous capture of ST40 branch trace information using the ST Micro Connect. See *Section E.2.1: Continuous capture support* for details of the options to this command.
`reset`		Reset the branch trace buffer contents.
`save`	*`file`* `[reset]`	Write the branch trace to *`file`*. The file is written in the same format as the `display` command. If `reset` is specified, then save the initial branch trace contents instead of the current contents.

Table 69. Branchtrace subcommands

Subcommand	Options	Description
`savesim`	*`file`* `[reset]`	Write the branch trace to `file` in the same format as the simulator `sim_branchtrace` command (see *sim_branchtrace command on page 108*). If `reset` is specified, then save the initial branch trace contents instead of the current contents.
`status`		Display the configuration of the branch trace buffer.

When using the `append`, `appendsim`, `display`, `save` or `savesim` subcommands, use the `reset` option to report the data held in the branch trace buffer immediately after connecting to the target with GDB. This can be useful as a post-mortem debugging aid when reconnecting to a target after a crash.

E.2.1 Continuous capture support

Use the following command to configure, enable and disable continuous capture of ST40 branch trace information using the ST Micro Connect:

```
branchtrace remote size | on | off | stopon | stopoff
```

The options are as follows:

- *`size`* specifies the maximum number of branch trace records to be saved by the ST Micro Connect. If the number of captured records exceeds this figure, then the STMC discards the oldest records. The default size is 0.
- `on` enables the continuous capture of branch trace records. When this feature is enabled, the branch trace subcommands query the records captured by the STMC rather than querying the branch trace hardware directly. If *`size`* has not been specified, then no branch trace records are stored by the STMC and the target stops when the ST40 hardware trace buffer becomes full.
- `off` disables the continuous capture of branch trace records. Any records saved by the STMC are discarded. After a `branchtrace remote stop` command, the branch trace subcommands revert to querying the ST40 branch trace hardware directly.
- specify the option `stopon` to cause the target to stop when the STMC branch trace record buffer is full. When this occurs, GDB reports that the target has stopped due to a `SIGINT` (that is, as though the user has pressed **Ctrl + C** in the GDB command console). After restarting, the target will stop again when the STMC branch trace buffer is refilled (discarding its previous contents).
- specify the option `stopoff` to disable a previous `stopon` and revert to previous behavior; that is, to discard the oldest records in the branch trace buffer without stopping the target. This is the default.

Note: Enabling continuous capture of branch trace information is intrusive because the ST Micro Connect has to extract the branch trace records from the branch trace hardware whenever the hardware FIFO is filled.

E.3 Output format

The `display`, `save` and `append` subcommands report the details of the most recently taken branches, with the details of each branch formatted as follows:

```
#index to   address in function [at location]
       from address in function [at location]
       Mode: type [opcode]
```

where:

index This is the record number in the range 0 to *n* , where 0 is the most recent branch performed by the target, *n* the oldest.

address This is the address of the branch source or destination.

function This is the name of the function at the given address. `??` indicates that the function name is unknown.

location This is the source location of the given address, if known.

type This is the class of branch being recorded, and can be `gn`, `sb`, `ex` or `N/A` (if the class is unknown, or if opcode decoding is switched off with the `decode off` subcommand). See *Table 68 on page 277* for information on traceable branch classes.

opcode This is the mnemonic of the branch instruction or (if this is not a branch instruction) a hexadecimal number. The report displays `0xffff` if opcode decoding is switched off.

Note: If the `reset` option of either the `append`, `appendsim`, `display`, `save` or `savesim` subcommands is specified, the `type` and `opcode` fields are not reported.

Figure 38 shows an example of a branch trace report.

Figure 38. Example branchtrace output

```
#0 to   0x880017a4 in main () at hello.c:7
   from 0x8800186c in _puts_r () at ../../src/newlib/libc/stdio/puts.c:95
   Mode: sb [RTS]
#1 to   0x8800185a in _puts_r () at ../../src/newlib/libc/stdio/puts.c:94
   from 0x88001d96 in __sfvwrite_r () at ../../src/newlib/libc/stdio/fvwrite.c:271
   Mode: sb [RTS]
#2 to   0x88001d80 in __sfvwrite_r () at ../../src/newlib/libc/stdio/fvwrite.c:270
   from 0x88001d6a in __sfvwrite_r () at ../../src/newlib/libc/stdio/fvwrite.c:264
   Mode: gn [BTS]
#3 to   0x88001d5a in __sfvwrite_r () at ../../src/newlib/libc/stdio/fvwrite.c:257
   from 0x88002c8a in _fflush_r () at ../../src/newlib/libc/stdio/fflush.c:208
   Mode: sb [RTS]
#4 to   0x88002c7e in _fflush_r () at ../../src/newlib/libc/stdio/fflush.c:208
   from 0x88002c18 in _fflush_r () at ../../src/newlib/libc/stdio/fflush.c:208
   Mode: gn [BRA]
#5 to   0x88002c18 in _fflush_r () at ../../src/newlib/libc/stdio/fflush.c:208
   from 0x88001e1c in __libc_lock_release_recursive () at ../../src/newlib/libc/sys/lock.c:53
   Mode: sb [RTS]
#6 to   0x88001e14 in __libc_lock_release_recursive () at ../../src/newlib/libc/sys/lock.c:51
   from 0x88002c14 in _fflush_r () at ../../src/newlib/libc/stdio/fflush.c:206
   Mode: sb [JSR]
#7 to   0x88002c12 in _fflush_r () at ../../src/newlib/libc/stdio/fflush.c:206
   from 0x88002c78 in _fflush_r () at ../../src/newlib/libc/stdio/fflush.c:206
   Mode: gn [BRA]
```

In *Figure 38*, the most recent branch (index `0`) is an RTS from `0x8800186c` in the function `_puts_r()` (found in `puts.c` line 95) back to `0x880017a4` in `main()` (`hello.c` line 7).

E.4 ST Micro Connect configuration options

If continuous capture support is enabled, the ST Micro Connect collects the branch trace data. Continuous capture can be configured using the `branchtrace remote` command (see *Section E.2.1: Continuous capture support*) or by issuing ST Micro Connect configuration options.

Table 70 lists the ST Micro Connect configuration options and their equivalent `branchtrace remote` commands.

Table 70. STMC configuration options

Configuration option	Equivalent to
`stmcconfigure branchtrace=on`	`branchtrace remote on`
`stmcconfigure branchtrace=off`	`branchtrace remote off`
`stmcconfigure branchtrace=reset`	`branchtrace reset`
`stmcconfigure branchtrace.stop=on`	`branchtrace remote stopon`
`stmcconfigure branchtrace.stop=off`	`branchtrace remote stopoff`
`stmcconfigure branchtrace.size=`*`size`*	`branchtrace remote `*`size`*

Appendix F ST TargetPack plugin

The ST TargetPack plugin provides the following services to GDB.

- It defines the memory mapped registers specified for an SoC by the ST TargetPack as GDB convenience variables.
- It defines GDB commands that can be used for displaying the contents of the memory mapped registers in various formats.

The convenience variables and GDB commands are similar to those generated by the `--gdb-mmrs` option of the **sttpdebug** tool provided in the ST Micro Connection Package. See *ST TargetPack user manual* (8020851) for information about the **sttpdebug** tool.

These features are accessible through the GDB `targetpack` command.

F.1 The targetpack command

When a host connects to a target using an ST TargetPack, the `targetpack` command is enabled automatically. It can also be enabled by issuing the GDB command `enable_targetpack` (as defined in the `targetpack.cmd` GDB command script.)

The `targetpack` command has the following format:

```
targetpack subcommand options
```

This command controls the ST TargetPack function specified by *`subcommand`* and *`options`*.

The subcommands supported by the `targetpack` command are listed in *Table 71*.

Table 71. Targetpack subcommands

Subcommand	Options	Description
`help`	[*`subcommand`*]	Display help for the `targetpack` command. If subcommand is specified, then more detailed help for *`subcommand`* is displayed.
`import`	*`targetstring`*	Import the ST TargetPack register set associated with the specified *`targetstring`*. If connecting to a target using the `sh4tp` (or related) connection command, then the command `targetpack import` is automatically invoked after connecting to the target.
`export`	[*`file`*]	Export the register convenience variables and commands into GDB. To export the convenience variables and commands to a GDB command script for later use, specify the name of the file with *`file`*.

For example, when connected to a target, the following command sets up the memory mapped user commands and convenience variables:

```
targetpack export
```

This command also displays further information on how to list the available memory mapped register names.

The following examples show how to use the memory mapped user commands and convenience variables.

List all register groups user commands:

```
help mmrs_component
```

Display the contents of the ST40 CCN peripheral registers:

```
mmrs_CCN
```

List all registers:

```
help mmrs_register
```

Decode and display the register contents of entry 0 in the PMB address array:

```
mmrs_pmb_addr_array_entry0 -v
```

List all register convenience variables:

```
help mmrs_convenience
```

Display the ST40 cache control register:

```
p/x *$mmrs_CCN_CCR
```

The `targetpack` command is available only when connected to a target. However, it is not necessary to connect to a target using an ST TargetPack in order to to use the `targetpack` command, nor is it necessary to import or export the same *`targetstring`* used for the original target connection.

The following example illustrates this. After connecting to a simulated MB448 target, use the `targetpack` command to export the corresponding ST TargetPack register set to a GDB command script called `mb448regs.cmd`:

```
mb448sim
enable_targetpack
targetpack import stmc:mb448:st40
targetpack export mb448regs.cmd
```

Appendix G Simulator configuration variables

The ST40 simulator can be configured either by setting configuration variables individually, or by loading a file that defines the variables to set as a series of `+variable=value` statements. Both the functional simulator and the performance simulator can be configured using the variables listed in *Table 72*.

Table 72. Common configuration variables

Variable	Description	Default
`cprc.disable_ovf_setting`	If true, disable the WDT overflow flag so that timing of interrupts can be controlled externally.	false
`cprc.pclks_per_iclk`	The ratio of `pclk` to `iclk` used for WDT.	2
`cpu.allow_pmb_reset`	Allow unmapped PMB reset.	false
`cpu.branch_trace.queue_size`	Number of addresses that can be stored in the branch trace queue. Can be any value in the range of 16 to 100000000, and must be a multiple of 2.	16
`cpu.branch_trace.skip_empty`	When true reads from the trace buffer skip any empty entries.	false
`cpu.ccr_emode_initial_value`	Select initial value of EMODE (emulation mode) field in the CCR (cache control register). Set to true for 2-way caches and false for direct mapped caches. This variable is only active when the family type is FPU.	false
`cpu.ccr_emode_read_only`	Set to true to make the EMODE field of the CCR read only. This variable is only active when the family type is FPU.	false
`cpu.csp_base_address`	Programmable CSP base address (must be 2K aligned).	`0xFFC00000`
`cpu.dcache.nr_partitions`	Number of partitions (set size) in the data cache.	2
`cpu.dcache.lines_per_part`	Number of cache lines per partition.	512
`cpu.dcache.cache_line_size`	Number of bytes in a cache line.	32
`cpu.dcache.perfect`	Set to true to get perfect caching (that is, always hit the cache).	false
`cpu.debug_interrupts`	Set to true to enable debug message printing for interrupt launch.	false
`cpu.debug_lsu`	Set to true to display debug info relating to load/stores	false
`cpu.default_imprecise_trap_delay`	Sets the default delay (as an instruction count) for imprecise traps.	0
`cpu.disable_esp`	Set to true to disable emulation support peripheral.	false
`cpu.dynacon.enable`	Set to true to enable the memory region.	true
`cpu.dynacon.base`	Base address of the memory region.	`0xFFBFFF00`
`cpu.dynacon.size`	Size of the memory region in bytes.	256
`cpu.enable_experimental_insns`	Set to true to enable experimental instructions.	false

Table 72. Common configuration variables (continued)

Variable	Description	Default
`cpu.enable_sh4_mp_isa`	Set to true to enable multi-processor mode.	false
`cpu.enable_sh4_300_isa`	Set to true to enable the ST40-300 series ISA.	false
`cpu.force_pvr_value`	Override default PVR value when non-zero.	0
`cpu.force_cvr_value`	Override default CVR value when non-zero.	0
`cpu.reset_address_offset`	Programmable reset address.	`0xA0000000`
`cpu.icache.nr_partitions`	Number of partitions (set size) in the instruction cache.	2
`cpu.icache.lines_per_part`	Number of cache lines per partition.	256
`cpu.icache.cache_line_size`	Number of bytes in a cache line.	32
`cpu.icache.perfect`	Set to true to get perfect caching (that is, always hit the cache).	false
`cpu.propagate_undefined_values`	When set to true, if any source operand is undefined, then the simulator makes the corresponding output operand undefined also.	false
`cpu.sh4_family`	Select processor family: 0: FPU (processor includes floating point unit) 1: MPU (processor excludes floating point unit) 2: MCU (processor excludes floating point unit and memory management unit)	0
`cpu.sh7751_features`	Set to true to enable SH7751 specific behavior, such as store queue behavior.	false
`cpu.support_cache_index_modes`	Set to true to enable support for cache index modes. This variable is only active when the family type is FPU.	true
`cpu.ub_bits_reset_value`	Reset values of physical address space control register field PASCR.UB[0..7] .	false
`rtc_rate`	Ratio of `SysCLK` to `RTC` clock rate.	12207
`tmu.disable_unf_setting`	Used to disable setting of the TMU underflow flag so timing of interrupts can be controlled externally.	false

In addition to the variables listed in *Table 72* above, the performance simulator can be configured using the variables listed in *Table 73*.

Table 73. Configuration variables for the performance simulator

Variable	Description	Default
`cpu.allow_branch_target_pairing`	If true, branch can be paired with target instruction when issuing instructions.	false
`cpu.branch_cache.associativity`	Associativity of the branch cache. Permitted values are 1, 2, 4, or 8.	1

Table 73. Configuration variables for the performance simulator (continued)

Variable	Description	Default
`cpu.branch_cache.cache_type`	Number of states of branch history in each entry. 0: bi 1: tri 2: quad	tri
`cpu.branch_cache.entries`	Number of entries in the branch target cache. Permitted values are 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 or 1024.	512
`cpu.branch_cache.reduce_dont_evict_strong_entries`	When set to true, branches do not evict strong entries but strength reduce them instead.	true
`cpu.branch_cache.replacement`	Replacement policy of branch cache. 0: lru (least recently used) 1: lrr (least recently replaced)	lru
`cpu.branch_cache.write_blocks_read`	Set to true if updates prevent a lookup (for example, for a RAM implementation).	false
`cpu.cache_store_on_fill`	Set to true to allow cache write hits to take place while a cache line fill is pending.	false
`cpu.delayed_branches_first`	If true, delayed branches must come first in the issue pair.	false
`cpu.fetch.fetch_delay`	Number of cycles it takes to process an icache fetch request. Value is in the range of 1 to 3.	1
`cpu.fetch.fetch_queue`	Size of the fetch queue in number of opcodes.	8
`cpu.fetch.fetch_width`	Number of bytes fetched per request. Permitted values are 4, 8, 16, or 32.	8
`cpu.force_single_issue`	Set to true to force single issue of all instructions.	false
`cpu.forward_to_vector_opds`	If true, enables forwarding to vector operands.	false
`cpu.mach_forwarding_stage`	Define the pipeline stage when multiply MACH result is forwarded. 0: write_back 1: data_access 2: execute3	write_back
`cpu.macl_forwarding_stage`	Define the pipeline stage when multiply MACL result is forwarded. 0: write_back 1: data_access 2: execute3	write_back
`cpu.one_cyc_reg_move`	Set to true to make the latency of register to register moves 1 cycle (instead of 0).	false
`cpu.one_cyc_const_move`	Set to true to make the latency of constant to register moves 1 cycle (instead of 0).	false
`cpu.optimise_zero_disp_branch`	If true, implement zero displacement branches as a conditional instruction.	true
`cpu.record_rts_in_branch_cache`	If true, RTS instructions are recorded in branch cache.	true

Table 73. Configuration variables for the performance simulator (continued)

Variable	Description	Default
`cpu.return_stack`	Specifies depth of return stack used to optimize subroutine returns. Permitted values are any number between 0 and 16 inclusive.	0
`cpu.serialise_pr_load`	If true, serialize execution of PR load instruction (rather than implement interlock)	false
`cpu.skew_ex_to_e2`	If true, simple EX class instructions (that is, instructions with no additional resource usage) run in the E2 stage of the pipeline.	false
`cpu.store_depcheck_in_e3`	If true, perform a late dependency check on stored values in the E3 stage of the pipeline.	false
`cpu.store_writeback_delay`	Specifies delay (in cycles) after writeback store before dependent loads can progress from the MA (memory access) stage. Permitted values are any number between 1 and 5.	1
`cpu.take_br_class_branches_in_e1`	When false, BR resource usage branches are taken in the decode stage, when true they are executed in the E1 stage.	false
`cpu.tbit_too_late_to_branch`	If true, T bit arrives too late in decode to branch. (This variable is only relevant if branching from decode.)	false
`cpu.use_branch_cache`	If true, use the branch cache.	false
`cpu.use_instruction_freelist`	Set to true to use a free list of instruction states to improve performance.	true
`cpu.use_long_pipe`	Set to true to enable a three stage integer execute pipe and a six stage FPU execute pipe.	false
`use_tobu`	Set to true to use the ST40-100 series type bus arbitration unit.	false

Appendix H GDB os21_time_logging user command

OS21 records the elapsed time that a task has been run on the CPU. This information is available to an application by using the OS21 `task_status()` API.

As a convenience, the GDB `os21_time_logging` command is provided to display the task list with the elapsed time for each task. This command is defined in the GDB command script `os21timelog.cmd` and displays the information with the following format:

```
task-number [task-name] = time-usus (time-ticks ticks) [*]
```

where:

`task-number` is the OS21 task number

`task-name` is the OS21 task name

`time-us` is the elapsed time in microseconds

`time-ticks` is the elapsed time in clock ticks

`*` indicates the current task

For example:

```
(gdb) source os21timelog.cmd
(gdb) os21_time_logging
1 [Root Task] = 14607us (22824 ticks) *
2 [Idle Task] = 9985us (15602 ticks)
3 [task0] = 19995us (31243 ticks)
4 [task1] = 39994us (62491 ticks)
5 [task2] = 59992us (93738 ticks)
6 [task3] = 79993us (124990 ticks)
```

Note: As the CPU clock is still running when the target is under the control of GDB, this time will be accumulated against the current task (indicated by a `*`) when the target is restarted. Using the same example as above but having previously already hit a breakpoint in `Root Task`:

```
1 [Root Task] = 204545us (319602 ticks) *
2 [Idle Task] = 9985us (15602 ticks)
3 [task0] = 19994us (31242 ticks)
4 [task1] = 39985us (62478 ticks)
5 [task2] = 59993us (93740 ticks)
6 [task3] = 79992us (124988 ticks)
```

The time in each task is comparable except for `Root Task` which now includes the time accumulated while the target was under the control of GDB.

Revision history

Table 74. Document revision history

Date	Revision	Changes
10-Jul-2015	23	– Added information about MTT (multi-target trace) in *Section 1.1: Toolset features on page 14*, *Section 1.3.2: Libraries on page 20* and *Section 1.5.3: The examples directory on page 28*.
09-Jul-2013	22	Throughout: minor revisions and corrections. Throughout: removed all references to **sh4gdbtui** and s**h-superh-elf-gdbtui**. as these are no longer distributed with the ST40 micro toolset. – Update to *Section 1.1: Toolset features on page 14* to include mention of profile and coverage tools. – Removed references to the deprecated GDB command `regs` in *Section 4.2.1: Using GDB on page 54*. – Updates to *Table 36: Errors returned by rl_errno() on page 153*. – Updates to *Section 11.5.5: Importing and exporting symbols on page 158*. – Updates to *Section 11.5.6: Optimization options on page 159*. – Updates to examples throughout *Appendix A: Toolset tips on page 223*. – Updates to *Section B.1.5: Link time optimization on page 253*.

Table 74. Document revision history (continued)

Date	Revision	Changes
20-Nov-2012	21	Throughout: minor revisions and corrections. – Added *ST Micro Connection Package documentation suite on page 12* – Update to *Section 1.4.7: Threading on page 25*. – Update to *Table 6: sh4gcc command line quick reference on page 34* – Update to *Table 7: sh4gcc SH-4 specific options on page 36* – Update to *Table 8: sh4gcc SuperH configuration specific options on page 37*. – Updated *Section 4.1.1: Using an ST TargetPack on page 50*. – Updated *Section 4.1.3: Using a GDB script with an STMC2 on page 51*. – Added *Section 4.1.5: Identification of the STMCLite on page 53*. – Update to *Section 5.1: Getting started with STWorkbench on page 68* – Update to *Table 16: Additional sh4gdb commands (not SuperH specific) on page 62*. – Update to *Table 42: sh4gcc linker options to enable OS21 Trace on page 172*. – Added *os21_trace_destructor_user on page 205* and *os21_task_trace_destructor_user on page 206*. – Update to *Section 12.15: Trace always on on page 212*. – Updates to examples in *Section A.6: Debugging with OS21 on page 231* – Update to *Section A.11: Using Cygwin on page 243* – Added *Section B.1.5: Link time optimization on page 253* – Updated *Section B.2.2: ST TargetPack on page 262* – Added *__builtin_trap() on page 265*.
12-Oct-2011	20	Minor changes to formatting. Update to *Section 4.1.4: Auto-detect connection on page 53* relating to auto-detection of STMC type. Minor update to *Section 11.5.5: Importing and exporting symbols on page 158* relating to the names in the symbol list Minor update to *Table 56: SH-4 specific GCC options on page 247* relating to -mieee option. Added *Section B.1.6: Stack overflow checking on page 256*. Revised *List of API functions on page 301* and *Index on page 305*. Added *List of built-in GDB commands on page 302* and *List of GDB user commands on page 303*.

Table 74. Document revision history (continued)

Date	Revision	Changes
5-Oct-2010	S	Throughout: minor revisions and corrections. Throughout: references to ST Micro Connect Lite added where appropriate. Amended list of supported platforms in *Section 1.1: Toolset features on page 14* and *Section 7.2: Requirements on page 93*. Amended the location of the `boardspecs` file in *Section 3.5.1: GCC board support setup on page 42*. Amended the location of the `runtimespecs` file in *Section 3.6.1: GCC run-time support setup on page 47*. Added *Section 4.1.4: Auto-detect connection on page 53*. Amended syntax of `disassemble` command in *Table 15: sh4gdb command quick reference on page 60*. Added `${variable}` to *Table 16: Additional sh4gdb commands (not SuperH specific) on page 62*. Amended the location of the `boardspecs` file in *Section 9.3: Adding support for new boards on page 131*. Modified specification of Trace user definition file in *Section 12.1.2: User definition file on page 165*. Added new tool os21usertracegen in *Section 12.1.3: os21usertracegen host tool on page 168* and *Section 12.1.4: os21usertracegen example on page 171*. Removed Section 12.6.3: Tips for creating an os21usertrace definition file.
30-Nov-2009	R	Throughout: minor revisions and corrections. Added *Passing arguments from environment variables on page 19*. Added a reference to `version.txt` in *Section 1.5: Release directories on page 26*. Revised *Section 3.5.1: GCC board support setup on page 42* and *Section 3.6: Run-time support on page 46*. Revised *Section 4.2.3: Connecting to a running target on page 58*. Added previously undocumented tutorials to *Chapter 5: Using STWorkbench on page 68*. Added *Section 8.3.4: Commands in shsimcmds.cmd on page 108*. Added details of “OS21 Trace user record” throughout *Chapter 12: OS21 Trace on page 163*. Revised *Section A.11: Using Cygwin on page 243*. Added *Impact of the -m4 option on the assembler on page 249*. Added several new `branchtrace` commands in *Appendix E: Branch trace buffer on page 277*. Added *Section E.2.1: Continuous capture support on page 279*. Added *Section E.4: ST Micro Connect configuration options on page 281*. Corrected `os21_time_log` to `os21_time_logging` in *Appendix H: GDB os21_time_logging user command on page 288*.

Table 74. Document revision history (continued)

Date	Revision	Changes
21-May-2009	Q	Throughout: minor revisions in style and layout to ensure closer alignment with *ST200 toolset user manual* (8063762) Throughout: minor revisions and corrections. Added *Section 1.4.5: The zlib library on page 24*. Changed section numbering in *Chapter 4: Cross development tools on page 50*, promoting some level 3 headings to level 2. Modified *Section 4.1.1: Using an ST TargetPack on page 50*. Added *Section 4.2.3: Connecting to a running target on page 58* to this manual in order to bring it into alignment with the layout of the *ST200 toolset user manual* (8063762) Added continue-after-exit subcommand to *Table 18: Subcommands available with the set shtdi and show shtdi commands on page 64*. Amendments made to *Table 19: sh4xrun command line options on page 65*. Removed menus from *Chapter 6: Using Insight on page 75*. Updated *Chapter 12: OS21 Trace on page 163*. Added *OS21 profiler initialization and start on page 217*. Added *OS21 profiler cancel on page 218*. Added two additional commands to *OS21 profiler status reporting on page 218*. Revised instructions to manage a memory partition in *Section A.1: Managing memory partitions with OS21 on page 223*. Revised table of permitted frequencies for an ST Micro Connect 1 in *Section A.9: Changing ST40 clock speeds using GDB command scripts on page 240*. Added *Compiling libraries for a specific core on page 249*. Added details relating to sh4gprof to *Section D.5: Examples on page 276*. Added *Appendix H: GDB os21_time_logging user command on page 288*. Revised *Index* and created a separate *List of API functions on page 301*.

Table 74. Document revision history (continued)

Date	Revision	Changes
26-Nov-2008	P	Throughout: minor revisions and corrections. Updated feature differences between short form and long form versions of the tools in *Section 1.3: Distribution content on page 17*. Added *Section 2.2: OSPlus on page 33* to provide a brief overview of OSPlus. Added note relating to linker symbols in *Section 3.5.1: GCC board support setup on page 42*. Added note about updating STMC firmware in *Chapter 4: Cross development tools on page 50*. Removed all references to Solaris in *Chapter 7: Building open sources on page 93*. Added note relating to Cygwin **make** in *Section 7.4: Building the packages on page 96*. Added *Section 12.8: Structure of trace binary files on page 181*. Added *Chapter 13: Dynamic OS21 profiling on page 214*. Added footnote to *Table 65: perfcount subcommands on page 269*. Added `reset` option to relevant commands in *Section E.2: The branchtrace command on page 278*. Added decode subcommand in *Section E.2: The branchtrace command on page 278*. Added *Section E.3: Output format on page 280*. Added *Section D.2: Trace profiler output format on page 273* and *Section D.3: Range profiler output format on page 274*. Added *Appendix F: ST TargetPack plugin on page 282*.

Table 74. Document revision history (continued)

Date	Revision	Changes
12-Jun-2008	O	Throughout: removed board specific information (several large multipage tables) and placed this information in HTML files to be distributed with the toolset. Added new syscalls functions in *Section 1.4.6: The syscalls low-level I/O interface on page 24*. List of OS21 examples has been updated in *Section 1.5.3: The examples directory on page 28*. Added the `-trace` options in *Section 3.1.2: GCC SuperH configuration specific options on page 37*. Added information on linking to multicore SoCs with ST200 cores in *Section 3.5.1: GCC board support setup on page 42*. Added several new **sh4gdb** configuration specific commands in *Section 4.2.6: Additional GDB commands on page 62*. Amended *Section 4.3: Using sh4xrun on page 65* in respect of the `-e` option. Added new command line options to **sh4xrun** in *Section 4.3: Using sh4xrun on page 65*. Some minor amendments to *Chapter 6: Using Insight on page 75*. Updated version number of the open source packages in *Section 7.1: Introduction to open sources on page 93*. Updated the list of examples in *Chapter 10: Booting OS21 from Flash ROM on page 135*. Moved *OS21 Trace* from Appendix G to *Chapter 12*. Added `-trace` option in *Chapter 12: OS21 Trace on page 163*. Added four new API functions in *Chapter 12: OS21 Trace on page 163*.

Table 74. Document revision history (continued)

Date	Revision	Changes
12-Dec-2007	N	Added several new examples in *Section 1.5.3: The examples directory on page 28*. Added new *Section 3.5.3: Alternative placement of sections on page 46* to describe a new example. Minor changes to *Chapter 4* in relation to ST MicroConnect 2. Added usage of `compare-sections` GDB command to *Table 15: sh4gdb command quick reference on page 60*. Changes to usage of `enable console`, `enable rtos` and `enable sharedlibrary` commands and added `set/show shtdi-wait-timeout` in *Table 17: SuperH configuration specific sh4gdb commands on page 63*. Added change to the usage of the `sh4xrun -c` command line option in *Table 19: sh4xrun command line options on page 65*. Major revision to *Chapter 7: Building open sources*. Removed obsolete OS21 configurable options from *Chapter 9: OS21 source guide on page 126*. Corrected minor errors in *Table 33: Examples of booting from Flash ROM on page 135*. Added new compile line option `-malign-small-blocks` in *Table 56: SH-4 specific GCC options on page 247*. Added extra information to Appendix *A* to assist users of **sh4objdump**. Added information on the use of precompiled headers in *Table A.12: Using precompiled headers on page 244*. Added `save` subcommand to the `branchtrace` command, in *Section E.2: The branchtrace command on page 278*. Added *Appendix D: Profiler plugin on page 270*. Added *Appendix G: Simulator configuration variables on page 284*. Added *12: OS21 Trace on page 163*. Throughout, removed misleading references to Super-H and SH-4. Corrected other minor, non-technical errors.

Table 74. Document revision history (continued)

Date	Revision	Changes
20-Mar-2007	M	Moved to new template. A new manual has been added to the ST40 documentation suite; *ST40 Micro Toolset GDB Command Scripts* (8045872). JTAG Control and ST Micro Connect setup appendices have been removed and incorporated into *ST40 Micro Toolset GDB Command Scripts* (8045827). GDB commands section has been removed and incorporated into *ST40 Micro Toolset GDB Command Scripts* (8045872). Legacy boards have been identified throughout. Updated, *Section 3.1: The GNU compiler (GCC) on page 34*. Updated, *Section 3.5: Board support on page 40*. Updated *Chapter 4: Cross development tools on page 50*. Updated *Section 8.3: ST40 simulator reference on page 104*. Updated *Section 9.2: Building the OS21 board support libraries on page 128*. Updated *Section 9.3: Adding support for new boards on page 131*. Updated *Section 10: Booting OS21 from Flash ROM on page 135*. Updated *A.5: Access to uncached memory on page 230*. Updated *A.7.3: Debugging OS21 boot from ROM applications on page 237*. Updated *A.9: Changing ST40 clock speeds using GDB command scripts on page 240*. Updated *B.2.1: Command scripts on page 262*. Updated *B.3.1: Default C run-time bootstrap on page 265*.
13-Sep-2006	L	In *Preface*, added *Acknowledgements on page 13*.

Table 74. Document revision history (continued)

Date	Revision	Changes
05-Sep-2006	K	Completely updated for R4.0.1 release of the toolset. Eclipse has been replaced with STWorkbench which is based on the Eclipse IDE. Updated the supported targets. In *Preface*, updated the documents included in the *ST40 documentation suite on page 11*. In *Toolset overview*, the SuperH configuration now provides support for the ST40-300 core. Updated *Threading on page 25*. Added details of the failsafe example in *OS21 examples on page 28*. Moved the installation instructions to the HTML pages on the CD. In *Introducing OS21*, OS21 now features user installable exception handlers. In *Code development tools*, updated the board support information in *Board support on page 40*. In *Cross development tools*, updated the *GDB command line reference on page 59* and the *Console settings on page 65*. In *Using STWorkbench*, renamed the chapter from Using Eclipse and updated to describe the STWorkbench. In *Using Insight*, updated *Using the Source Window on page 76*. In *Building open sources*, updated the version numbers for the list of open source packages delivered. Updated the requirements information. In *OS21 source guide*, updated the list of *Configuration options on page 126* and the example used in *Adding support for new boards on page 131*. In *Booting OS21 from Flash ROM*, added details of the failsafe example and updated the *Overview of booting from Flash ROM on page 136*. In *Toolset tips*, updated *Debugging with OS21 on page 231*. Updated *General tips for GDB on page 236*, including adding the sections *Power up and connection sequence on page 238* and *Using hardware watchpoints on page 239*. In *Development tools reference*, updated *Hardware watchpoint support on page 263*.
11-Oct-2005	J	Completely updated for R3.1 Product release of the toolset. Updated to GDB 6.3 and Insight 6.1. Added support for STb7109 and STb7100-Ref. In Introducing the ST40 Micro Toolset chapter: Added details of sh4gdbtui, the text user interface for the debugger. Updated details of the os21prof and sh4rltool tools. In Cross development tools chapter: Added details of sh4gdbtui, the text user interface for the debugger. Updated the list of GDB SuperH configuration specific options and SuperH specific GDB commands. The register40.cmd commands no longer require an additional argument for endianness. In Using Eclipse: Added new chapter to introduce Eclipse. In Building open sources chapter: Updated to provide more information for building sources, particularly on Windows. In Reloctable loader library chapter: The section on writing and building a relocatable library or main program has been updated.

Table 74. Document revision history (continued)

Date	Revision	Changes
24-Apr-2005	I	Completely updated for R3.0.3 Product release of the toolset. All chapters have been updated. The JTAG control appendix has been completed.
03-Mar-2005	H	Completely updated for R3.0.2 Beta release of the toolset. All chapters have been updated. The Relocatable loader library chapter and the Performance counters and Branch trace buffer appendices have been added. The JTAG control appendix has been added and will be completed for the product release. The Toolset changes since R2.0.5 appendix has been moved to the CD-ROM.
29-Sep-2003	G	Minor rephrasing and grammatical changes. Details of the ST220-Eval development board have been added. Replaced UDI with H-UDI. In Code development tools chapter: Updated list of recognized boards.
26-Sep-2003	F	Carried out minor rephrasing and grammatical changes. Added details of SH-4 202 support. Updated ST40 version number to 2.1.3. In Introducing the ST40 Micro Toolset chapter: Added details of the sti5528loader example. In OS21 source guide chapter: Replaced GDB_START and GDB_END with GDB_BEGIN_EXPORT and GDB_END_EXPORT in the GDB OS21 awareness support section. In Toolset tips appendix: Added the section Just in time initialization. In Toolset changes since R2.0.5 appendix: Changed OS21 version to V2.1.

Table 74. Document revision history (continued)

Date	Revision	Changes
11-Aug-2003	E	The entire manual has been restructured and the following new chapters and appendices have been added: – Code development tools, Cross development tools, Core performance analysis guide, Development tools reference, ST Micro Connect setup, Toolset changes since R2.0.5. – In addition the following changes have been made: – Added Windows XP to the supported Windows versions. Updated ST40 version number to 2.1.2. Updated GCC version number to 3.2.1. Replaced ST40RA166 with ST40RA. Renamed STLite/OS20 to OS20. In Preface: Added the section Conventions used in this guide. In Introducing the ST40 Micro Toolset chapter: Updated list of Libraries delivered and added subsections for the C library, C++ library and threading. Added descriptions of the new OS21 examples. In Using Insight chapter: Chapter has been rewritten. In OS21 source guide chapter: Updated the options described in the Configurable options section. Updated details of the support files in the Building the OS21 board support libraries section. In Booting OS21 from ROM chapter: Chapter has been rewritten. In Porting from OS20 chapter: Updated location of interrupt.h OS21 header file. Updated Interrupts and caches. In Toolset tips appendix: Added the section Memory managers. Managing critical sections in OS21: The section task / interrupt critical sections has been rewritten. Debugging with OS21: The examples have been replaced.
19-Aug-2002	D	Changed document title to ST40 Micro Toolset User’s Guide. Added the chapters Using sh4xrun, Using Insight, Building open sources and OS21 for ST40 source guide. Added the appendix, Toolset tips. Changed the term “include files” to “header files“. Added details of the simulator. In Preface: Added License information section. In Introducing the ST40 Micro Toolset chapter: Changed name from Introducing the GNU tools. Added details of where the GNU sources are located on the CD. Added footnote relating to big-endian versions of the libraries. Added introductory paragraph to the Installation section. Added sh4chess to list of tools. Expanded note explaining library locations in the Libraries delivered section. Changed the description of the getting started examples in the section, The examples directory. Corrected the location of the st40.bat file. Added details of Cygwin to the Windows installation description. Corrected the names of display40.cmd commands. Updated list of commands in sh4targets.cmd. Added how to access the help for the GDB command scripts.
20-Jun-2002	C	Added details of os21/soaktest example. Corrected ST40RA166 Overdrive board to ST40STB1-ODrive board. Replaced minor typing and grammatical errors.

Table 74. Document revision history (continued)

Date	Revision	Changes
24-May-2002	B	Removed details of unsupported boards. Updated version numbering. Added Index. In Introducing the GNU tools chapter: Added definition of bare machine application. Updated details of the documents directory. Added footnotes for boards that are no longer in production. Replaced board codes with full production names. Removed references to the bare library directory. Updated the Release directories section. Renamed connect.cmd to sh4si.cmd.
12-Mar-2002	A	Initial release.

List of API functions

List of built-in GDB commands

List of GDB user commands

Index