ST20 Embedded Toolset R1.9 User Manual

ADCS 7143840G

November 2001

The C compiler implementation was developed from the Perihelion Software “C” Compiler and the Codemist Norcroft “C” Compiler.

The C++ language front-end was developed under a Licence Agreement between EDISON DESIGN GROUP, Inc (EDG) and STMicroelectronics (formerly SGS-THOMSON Microelectronics) Limited.

This product incorporates innovative techniques which were developed with support from the European Commission under the ESPRIT Projects:

- P2701 PUMA (Parallel Universal Message-passing Architectures)
- P5404 GPMIMD (General Purpose Multiple Instruction Multiple Data Machines).
- P7250 TMP (Transputer Macrocell Project).
- P7267 OMI/STANDARDS.
- P6290 HAMLET (High Performance Computing for Industrial Applications).
- P606 STARLIGHT (Starlight core for Hard Disk Drive Applications).

STMicroelectronics GROUP OF COMPANIES
Australia - Brazil - Canada - China - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan
Malaysia - Malta - Morocco - Singapore - Spain - Sweden - Switzerland - United Kingdom - U.S.A.

http://www.st.com

Document number: ADCS 7143840G (Previously identified by the series 72-TDS-505-xx.)

Contents

Preface

This manual is the user manual for the R1.9 release of the ST20 Embedded Toolset, which can be run on the following hosts:

- PC running Windows 95/98/2000/NT,
- PC running Red Hat Linux V6.2;
- Sun 4 running Solaris 2.5.1 or later.

Note: the ST Visual product is only available on Windows and Solaris platforms. The EMI configuration tool `stemi` is available on Windows, see the Delivery Manual for licensing details.

About this document set

The document set provided with the toolset comprises:

- **ST20 Embedded Toolset Delivery Manual (ADCS 7257995)** - provides installation instructions, a summary of the release and a list of changes since the previous revision.
- **ST20 Embedded Toolset User Manual** (this manual). The User Manual is split into three parts:
  - Part 1 presents introductory material comprising an overview of the toolset and a ‘getting started’ guide.
  - Part 2 describes how the core features of the toolset are used to build and run application programs. The guide includes compiling and linking, connecting to a target, loading programs, application debugging and ROM systems.
  - Part 3 is a user guide for the STLite/OS20 real-time kernel. This part also starts with an overview and ‘getting started’ and then contains separate chapters for each of the main features supported, that is the kernel, memory and partition management, tasks, semaphores, message queues, real-time clocks, interrupts, as well as the device, cache and core-specific functions.
- **ST20 Embedded Toolset Reference Manual (ADCS 7250966)** - the reference manual is divided into five distinct parts:
  - ‘Advanced facilities’ describes each of the support tools provided with the toolset, for example, an assembler, EMI tool, librarian, lister, and ST20 instruction set simulator. It also describes facilities such as code and data placement, the stack depth and memory map files, the use of relocatable code units, profiling and trace facilities, using `st20run` with STLite/OS20 and the advanced configuration of the STLite/OS20 kernel.
  - A language reference describes how the ANSI C and C++ languages have been implemented by the toolset and provides details of the toolset’s preprocessing facilities.

   - A library reference provides definitions of the toolset libraries.
   - An STLite/OS20 reference provides definitions of the STLite/OS20 real-time kernel functions.
   - A complete command language reference describes the command language shipped with the toolset and provides an alphabetical list of command definitions.
- **ST Visual Documentation Set** - provides the following titles:
   - ST Visual Make Getting Started Guide
   - ST Visual Make User's Guide
   - ST Visual Other Tools

Conventions used in this manual

The following typographical conventions are used in this manual:

Bold type	Used to denote special terminology, for example register or pin names.
`Teletype`	Used to distinguish command options, command line examples, code fragments, and program listings from normal text.
Italic type	In command syntax definitions, used to represent an argument or parameter. Used within text for emphasis and for book titles.
Braces {}	Used to denote a list of optional items in command syntax.
Brackets []	Used to denote optional items in command syntax.
Ellipsis `...`	In general terms, used to denote the continuation of a series. For example, in syntax definitions denotes a list of one or more items.
|	In command syntax, separates two mutually exclusive alternatives.
	A change bar in the left margin, indicates a change from the previous version of the manual. This may indicate a change in the functionality of the toolset or merely an updated description.

Command line conventions

Example command lines and directory path names are documented using UNIX/Linux style notation which makes use of the forward slash '/' delimiter. In most cases this should be recognized by Windows hosts, if not the forward slash should be substituted with a backslash character '\'. For example, the directory:

```
release/examples/simple
```

is the same as:

```
release\examples\simple
```

Command line options are prefixed by a '-' hyphen; this is Windows, UNIX and Linux compatible.

Examples of the debugging tools use the following convention to distinguish command prompts:

'%' is used to indicate the operating system command prompt, for example:

```
% st20run ....
```

'>' is used to indicate the interactive command language prompt, for example:

```
> break ....
```

Part 1 - Introduction

1 Toolset overview

The ST20 Embedded Toolset supports the programming and debugging of ST20 silicon devices with a Diagnostic Controller Unit (DCU) and the ST20-C1 or C2 processor cores.

1.1 Key features

- PC Windows 95/98/2000/NT, Sun 4 Solaris 2.5.1 and Red Hat Linux V6.2 hosted toolsets.
- Development tools and support libraries, include:
    - ANSI C compiler/linker driver tool `st20cc`, allowing the compilation and linking of multiple files with a single tool invocation;
    - Support for dynamically loadable code;
    - Assembler;
    - Librarian `st20libr;`
    - Lister tool `st20list;`
    - External memory interface configuration tool `stemi`, (Windows only);
    - ST20 instruction set simulator providing trace and cycle time statistics;
    - Full ANSI C library;
    - ST20-specific libraries for mathematics and debugging;
    - Bootstrap libraries;
    - C++ support tools and libraries.
- Extensive diagnostic facilities, include:
    - A combined loader and windowing debugger `st20run`, that enables:
        - Code to be loaded and run on ST20 silicon or simulator targets;
        - Interactive debugging, tracing and profiling;
        - Use of breakpoints and watchpoints for debugging;
        - The creation and display of instruction traces.
    - File I/O in diagnostic mode.
- STLite/OS20 Real-time multi-tasking kernel supports specific ST20-C1/C2 core features and facilitates portability of programs between ST20 platforms.
- A powerful and common command language interface provides:
    - Flexible control of linking;
    - Creation of debugging scripts;
    - Fine grain code and data placement.
- Two modes of booting:
    - Boot-from-ROM;
    - Boot via diagnostic controller (DCU) - diagnostic mode.

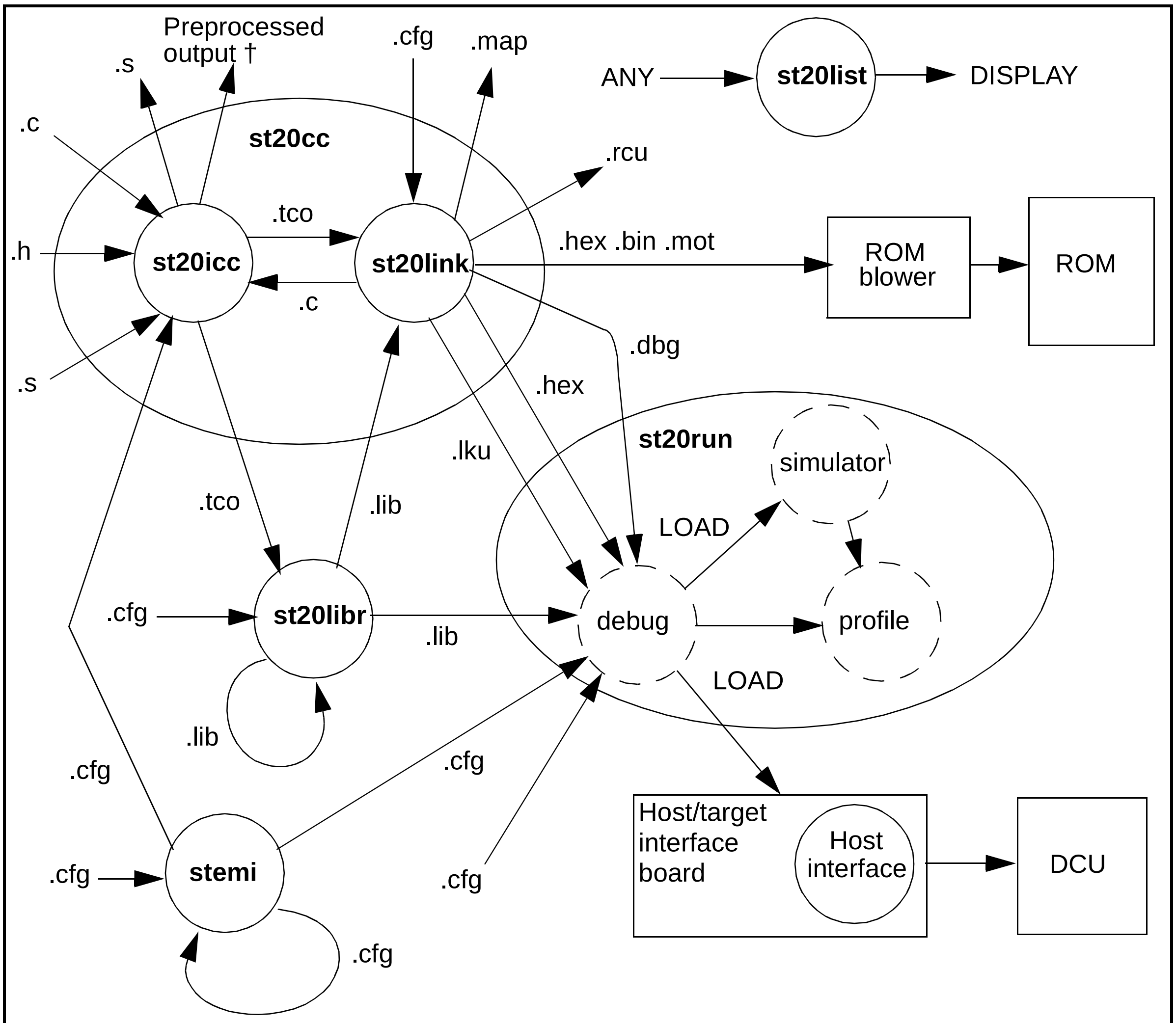
1.2 Toolset architecture

The ST20 Embedded Toolset architecture is described below in terms of developing a C program. C++ development is described in Chapter 4.

The ST20Embedded Toolset architecture for developing C programs can be divided into the following groups of tools:

- Tools to compile and link a program for blowing onto ROM, accessed via `st20cc`;
- Debugging tools for testing a program under development, accessed via `st20run` and `st20sim` the ST20 instruction set simulator;
- Support tools, such as the librarian and lister tools `st20libr` and `st20list`.

Figure 1.1 shows the overall architecture of the ST20 Embedded Toolset.



Note: File extensions are explained in Chapter 3.

† Preprocessed output is sent to standard out

Figure 1.1 ST20 Embedded Toolset architecture for ANSI C

1.2.1 st20cc driver tool

`st20cc` acts as a wrapper for an ANSI C compiler and a compacting linker. It provides a single tool interface to the compilation and linking phases of development, simplifying the command line and reducing the number of development steps.

`st20cc` takes C source and assembly code, compiles it and links it with any object or library file specified as input, to produce a file in one of the formats described in section 1.2.7.

`st20cc` is described in detail in Chapter 3.

1.2.2 st20run program load, run and debug tool

`st20run` is a multi-purpose diagnostic tool which enables applications to be loaded and run on either silicon or a simulator. It includes a debugger Graphical User Interface (GUI). `st20run` is described in detail in Chapter 7.

1.2.3 st20libr librarian

`st20libr` takes as input object files or existing libraries and combines them into a single library file which can then be passed to the linker. `st20libr` is described in detail in the "*ST20 Embedded Toolset Reference Manual*".

1.2.4 stemi configuration tool

`stemi` generates configuration values for ST External Memory Interface (EMI) devices. The tool requires a PC host running Windows and is described in the "*ST20 Embedded Toolset Reference Manual*".

1.2.5 st20list lister

`st20list` is a file lister. It can display most of the file types used by the toolset and has options to modify the display format. `st20list` is described in detail in the "*ST20 Embedded Toolset Reference Manual*".

1.2.6 st20sim simulator

`st20sim` is the ST20 Embedded Toolset instruction set simulator, which simulates a ST20-C1 or ST20-C2 processor core. It can generate trace data and cycle time statistics and may be invoked directly or as a target for `st20run`. `st20sim` is described in detail in the "*ST20 Embedded Toolset Reference Manual*". Access via `st20run` is described in Chapter 6 and Chapter 7.

1.2.7 Loadable file types

The toolset can produce loadable output in the following formats:

- ROM image files in one of a number of standard ROM image formats. A ROM image file is suitable for blowing directly into a ROM.
- Linked unit files, which can be loaded at run-time. These files are used during development when the overhead of producing ROMs is undesirable or when the final system architecture is not yet fixed. They can be loaded onto ST20 silicon or simulator targets.
- Relocatable code units that can be loaded dynamically at runtime.

All three output types can be debugged using `st20run`.

1.2.8 Diagnostic controller unit

ST20 parts include a Diagnostic Controller Unit (DCU). This is the interface for loading and debugging code. Toolsets versions R1.8.1 and earlier, only support parts with a DCU2. This toolset also supports ST20 parts (such as the STV0396 and the STi5514) that include the DCU3 which provides improved debugging facilities.

This toolset release does not make significant use of the new features that DCU3 provides over DCU2, but it supports all the tools' functionality provided in previous toolsets on both DCU2 and DCU3 based devices.

1.2.9 Host interface

The toolset supports connections from the following hosts:

- a Sun workstation running Solaris 2.5.1 or later;
- a PC running Windows 95/98/2000/NT;
- a PC running Red Hat Linux V6.2.

A hardware target will generally be an ST20 development board with a JTAG test access port (TAP) connected to an on-chip Diagnostic Control Unit (DCU). There are several methods of connecting the development board to the host:

- All hosts (Solaris, Windows and Linux) may be connected to the target hardware via Ethernet.
- PC Windows hosts may also be connected using the host parallel port.
- Windows 98 and Windows 2000 hosts may be connected to the target hardware using the host Universal Serial Bus port (USB).

Chapter 6 describes the various options for interfacing to the target hardware.

In the remainder of this document Solaris hosts are usually referred to as UNIX hosts.

1.3 Command language

A common command language is supported by the tools `st20cc`, `st20run`, `st20sim`, `st20libr` and `stemi`. Each of these tools uses a subset of the command language and certain key commands are recognized and used by more than one tool. The command language is described in the "*ST20 Embedded Toolset Reference Manual*".

Commands may be submitted to a tool:

- indirectly via a command file, this is a text file containing a list of commands and optional comments,
- or, in the case of the `st20run` debugging interface, interactively at the command line or via the debugging graphical interface.

In general, if a command is submitted which a particular tool does not use then the tool will ignore the command and proceed. **Note:** any exceptions to this are documented on a command basis in the "*ST20 Embedded Toolset Reference Manual*".

Command files may be specified on a tool command line by specifying the appropriate command line option; they may also be nested within other command files by using the `include` command.

The command language supports different types of commands, including:

- file access, (for example to include a named file);
- system definition, (for example to define the *target)*;
- system configuration, (for example to allocate memory regions);
- action commands, (for example to initialize a memory location or set a breakpoint.)

The term *target* is used to describe either the hardware or simulator that will run the application, or a definition of the hardware or simulator, including the processor type and memory.

1.3.1 Command procedures

It is very useful to group commands which perform a particular task into a command procedure, which may then be called as required. Command procedures are similar to C functions. Each procedure is given a name and executes one or more commands.

Procedures have the following format:

```
proc procedure_name {
    command list
}
```

For example, a command procedure might contain all the command language commands used to link and run an application on a particular target. Alternatively a collection of procedures might be written to define a number of targets.The `target` command which is used to specify a target to `st20run` requires a command procedure defining the target.

Within this manual, command procedures which are used to define, build or load a system are also known as '*configuration procedures*'. Typically they contain the commands which define the target and the application and are used as input to `st20cc`. They may also contain the commands which `st20run` uses to load code.

Command procedures may be grouped together into one or more command files also known as '*configuration files'*. It is good practice to give configuration files a standard file extension; the toolset uses the extension `.cfg`. Default configuration files for ST20 chip variants and evaluation boards are supplied with the toolset and may be used to run the supplied example programs. **Note:** user command files must not use the same names as the default command files supplied in the directory `$ST20ROOT/stdcfg`, see section 1.4.

Command procedures facilitate flexibility, enabling design definitions to be reused and shared. Systems may be easily reconfigured by calling a different command procedure and default platforms can be established by making a set of procedures globally visible, see section 1.4.1. The design process is simplified by having the ability to build up a single command procedure which may be input to each tool in the design cycle. Only those commands relevant to the particular tool will be executed.

Example

```
proc fax {
  chip STi5512
  memory EPROM 0x70000000 (16 * K) ROM
}
```

In this example the procedure `fax` defines an STi5512 chip with 16 Kbytes of EPROM starting at address `0x70000000`.

Calling command procedures

In order to call a particular command procedure, the procedure must first be defined. If it is defined in a command file, the file must first be made known to the tool. This is achieved by:

- specifying the command file on the command line, or
- referencing the command file with an `include` command:
  - from `st20run` in interactive mode;
  - from another command file which is then made known to the command line;
  - from a defaults file which is on the toolsets search path, as described in section 1.4.1.

The required command procedure can then be simply called by name, either from another command file or on the command line. For example:

```
st20cc hello.c -T hello.cfg -p phone
```

Where: `hello.c` is the application source file,

`-T hello.cfg` specifies a command file which contains:

```
proc phone {
.....
}
```

and `-p phone` calls the procedure `phone` which is defined in `hello.cfg`.

1.4 Toolset environment

Assuming the toolset is correctly installed, the following set -up should be complete:

- Your path should be set up to find the `bin` directory in the release directory.
- The environment variable `ST20ROOT` should be set to the root of the installation.
- A start-up script must be installed as described in section 1.4.1. This file contains some predefined command procedures, and is read by `st20cc`, `st20run`, `st20sim` and `st20libr`, on start-up. This file is necessary for the correct operation of the tools.
- The directory `stdcfg` should be installed in the release directory. This directory contains the default configuration files introduced in section 1.3.1.
- The directory `board` should be installed in the release directory. This directory contains target configuration files that will work on a selection of ST20 devices and evaluation boards.
- The directory `examples` should be installed in the release directory. This directory contains the example files supplied with the toolset.

Details of how to install the toolset and set up the toolset environment variables are included in the '*Delivery Manual*' which accompanies this release.

1.4.1 Start-up command language scripts

Up to three command language scripts are automatically executed on starting up the tools that support the command language: `st20cc`, `st20run`, `st20sim`, `st20libr` and `stemi`. On starting up, these tools will try to find and execute a script in each of three start-up command files. On a Sun running Solaris or PC running Linux, they will search for and execute the following start-up files in the following order:

1. `st20rc` in the directory named by the environment variable `ST20ROOT`;
2. `.st20rc` in your home directory (defined by the environment variable `HOME`);
3. `.st20rc` in the current directory.

On a PC running Windows, the tools will search as follows:

1. `st20rc` in the directory named by the environment variable `ST20ROOT`;
2. `st20.rc` in the directory named by the environment variable `HOME`;
3. `st20.rc` in the current directory.

Note: under Windows the `HOME` environment variable is set by the user, for example:

```
set HOME=C:\home
```

Under WindowsNT, if `HOME` is not set, then the home directory is derived from the system environment variables `HOMEDRIVE` and `HOMEPATH` as in the concatenation: `%HOMEDRIVE%%HOMEPATH%`.

The start-up command language script in the `st20rc` file in the toolset directory named by the environment variable `ST20ROOT` is supplied with the toolset and *care* must be taken *not* to overwrite or delete any of the commands or procedures supplied in this file. It is permissible for you to add commands and procedures to this file. Due to the order in which the files are executed it is also possible to overwrite values defined in the start-up file in the root directory file by values defined in the start-up files in either the home or current directory. Similarly values defined in the start-up file in the current directory may overwrite those defined in either the root or home directories.

Command language scripts placed in one of these files will be executed whenever `st20cc`, `st20run`, `st20sim`, `st20libr` or `stemi` starts up. This is a powerful facility for making command procedures globally visible and easy to share. Default command and configuration files should be placed in the appropriate start-up command file.

Example

```
# st20rc start-up file
include boardprocs.cfg
include targets.cfg
```

In this example the configuration file `boardprocs.cfg` is supplied with the toolset in the `board` directory. It contains configuration procedures for common ST evaluation boards.

1.5 Toolset version

The command `st20toolsetversion` can determine the toolset version for ST20 toolset version R1.9 and later releases. See the alphabetical list of commands in Part 5 of the "*ST20 Embedded Toolset Reference Manual*".

Toolset versions R1.6.2 to R1.8.1 can be determined by writing a procedure to probe the availability of particular commands in the toolset. An example procedure for identifying early toolset versions is provided in the `scripts` subdirectory within the `examples` directory supplied with the toolset.

2 Getting started with the tools

2.1 Introduction

This chapter gives step-by-step instructions on how to use the tools to compile and run a simple example program. It shows you how to:

- build a program as a linked unit (see section 1.2.7) and run it on silicon and the simulator;
- use configuration files to control the compile/link tool `st20cc` and the program load/debug tool `st20run`;
- use configuration files to place code and data in memory;
- share target definitions;
- create your own defaults;
- build and debug a ROM system.

The toolset must be installed before attempting the examples. Installation details for the toolset are included in the '*Delivery Manual*' which accompanies this release.

The examples described in section 2.2 through to section 2.5 build a linked unit using `st20cc` and then download it to the target using `st20run`. A linked unit is used for a rapid turnaround build-debug cycle, where the entire program is loaded from a host and executed out of RAM. The example given in section 2.6 describes how to generate a ROM image file. A ROM image file can be downloaded to the target by `st20run` or can run stand-alone.

2.1.1 Example files

A set of example files is provided with the release, in the `examples` subdirectory of the release directory. Copy the `examples` directory tree to a local working directory. The examples used in this chapter can be found in the `getstart` subdirectory of the `examples` directory.

The examples demonstrate the ease with which an application can be built and run on both the simulator and silicon. The examples are based around a STi5500 device which has a ST20-C2 core and runs on an EVAL-5500 demonstrator board. Modification of the examples to run on an alternative device or board is straightforward and is discussed in section 2.1.2.

The STi5500 has 4K of internal memory starting at 0x80000000, and the first 320 bytes are reserved for the processor's use. The EVAL-5500 board has 4 Mbytes of external DRAM starting at 0x40000000. A ROM generation example assumes a target with a further 512 Kbytes of ROM starting at 0x7FF80000. The memory map is shown in Figure 2.1.

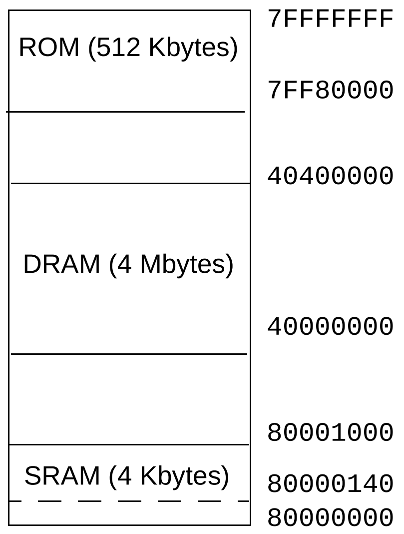


Figure 2.1 Memory map for ST20-C2 example

2.1.2 Using an alternative target

If you want to build a program to run on an alternative target you will need to change the configuration information that is passed to `st20cc` and `st20run`. The example file `st20tp3.cfg` contains the definitions for an ST20-TP3 device in an appropriate ST evaluation board and `st20dc1.cfg` contains the relevant definitions for an ST20-DC1 device. To use the ST20-TP3 in the above examples, simply replace instances of `sti5500.cfg`, `eval5500` and `eval5500sim` with `st20tp3.cfg`, `evaltp3` and `evaltp3sim` respectively.

In addition you may need to copy and edit the target definitions given in the supplied examples. Details of how to use the `target` command are given in Chapter 6.

2.1.3 Configuration

The use of configuration files and procedures is explained in Chapter 5. Commands are grouped together into procedures which together form a configuration file. When `st20cc` or `st20run` is invoked, a configuration file and a command procedure are specified on the command line and the tool will execute the specified command procedure, which may in turn call other command procedures.

The tools use a common target description together with other details which are specific to a particular tool. For example, `st20cc` needs to know details about the stack and heap requirements of the program, while `st20run` needs to know about the target connections, see Figure 2.2.

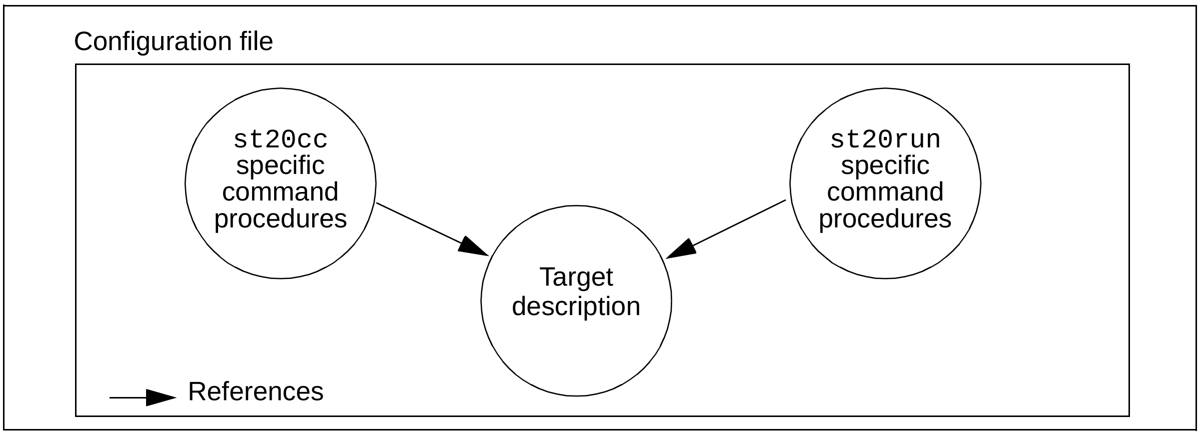


Figure 2.2 High-level overview of a configuration file

The toolset has a flexible approach to how command procedures are grouped within configuration files, so that each system can organize its configuration files to reflect its particular requirements.

2.2 Building and running with an STi5500 device

In practice you will need to use a target definition that represents your target system. The examples in this section target an STi5500 and use an appropriate ST evaluation board. In order to use this target, its description must be supplied to the tools.

The example consists of the following files, found in the `getstart` subdirectory of the `examples` directory:

- `hello.c` : Program source code.
- `sti5500.cfg` : Command file describing the target configuration.

2.2.1 Running the program on silicon

The development steps to build, run and debug the program are as follows.

1 Compile and link for an STi5500:

```
st20cc hello.c -T sti5500.cfg -p link -g
```

where:

`hello.c`	: The program to be compiled and linked.
`-T sti5500.cfg`	: Specifies the command file which describes the hardware and application configuration.
`-p link`	: Specifies the command language procedure to execute. This is found in `sti5500.cfg`.
`-g`	: Includes information for debugging

The command file `sti5500.cfg` specifies the target hardware and the memory requirements for the stack and heap of the program. The resultant linked unit file will be called `hello.lku`.

2 To run the program on an appropriate ST evaluation board, use the `st20run` command line as follows:

```
st20run -i sti5500.cfg -t eval5500 hello.lku
```

where:

`-i sti5500.cfg`	: Specifies the configuration file
`-t eval5500`	: Specifies the hardware target to run the program on.
`hello.lku`	: The linked unit to be loaded.

3 To debug the program with graphical user interface (GUI), use the `st20run` command line as follows:

```
st20run -i sti5500.cfg -t eval5500 hello.lku -g
```

When the debugger is invoked with a linked unit and the `-g` or `-d` option on the command line, a breakpoint is set on `main()` and the program is run until this breakpoint is hit. Step the program with the **Step** button (located on the ‘Code Window’) and note that the program’s output appears in the Xterm (Unix and Linux) or DOS (Windows) window.

Select **Exit: no save** in the File menu of the Code Window to close the debugger GUI.

Using an alternative target such as the ST20-TP3 is described in section 2.1.2.

2.2.2 Running the program on the simulator

The program may be run on the simulator by making some simple changes to the `st20run` command line:

1 Compile and link as in section 2.2.1.

2 On the `st20run` command line replace the target `eval5500` with `eval5500sim` as follows:

```
st20run -i sti5500.cfg -t eval5500sim hello.lku
```

3 To debug the program on the simulator, use the command line:

```
st20run -i sti5500.cfg -t eval5500sim hello.lku -g
```

2.3 Code and data placement example

This example uses a modified version of the simple `hello.c` which includes some data elements. It illustrates the use of the `place` command which changes the default placement of a program component in memory. The following example files are used and may be found in the `getstart` subdirectory of the `examples` directory:

- `helloplc.c` : Program source code.
- `sti5500.cfg` : Command file describing the application without placement.

In order to demonstrate the `place` statements we first build the program using the default memory mapping, then build the program with placement directives and compare the resultant memory mappings.

2.3.1 Building the linked unit

This is performed in the same way as in the example in section 2.2.1, except that an option is added to generate a map file for demonstration purposes:

1 Compile and link for an EVAL-5500:

```
st20cc helloplc.c -T sti5500.cfg -p link -g -M hello.map
```

where everything is as before except:

`-M hello.map` : Specifies the name of the map file.

2 Compile and link for a EVAL-5500 using placement directives:.

```
st20cc helloplc.c -T sti5500.cfg -p placelink -g -M place.map
```

where everything is as before except:

`-p placelink` :Link procedure with place commands
`-M place.map` :Specifies the name of the new map file.

3 Compare the map files:

`hello.map` : Shows default placements.
`place.map` : Shows placement using command in `place.cfg.`

Note the address of all the default sections and the symbols `main`, `bill` and `fred`.

Look at the procedure `placelink` in the file `sti5500.cfg`. This calls the procedure `link`, which we used for the example in section 2.1.1, and then places all default non zero-initialized data in internal memory, all default zero-initialized data (BSS) in internal memory and all code for the module `helloplc.c` in internal memory.

2.4 Sharing target definitions

In a multi-user environment it is useful to share target definitions so that all developers are working from a common base. This example shows how to achieve this. The example consists of the following files found in the `getstart` subdirectory of the `examples` directory:

- `hello.c` : Program source code.
- `sti5500.cfg` : Command file describing the application configuration.

In this example a configuration file is moved to a common area and this area is made known to the tools

2.4.1 Move to a common directory

Move `sti5500.cfg` to a common directory, such as `/st20tools/sharecfg`.

2.4.2 Start-up file

All the tools read a standard configuration file in your home directory on start-up. This file is called `.st20rc` for UNIX and Linux platforms or `st20.rc` for Windows platforms, (see section 1.4.1). Under Windows, the home directory is specified by the `HOME` environment variable.

An example start-up file called `st20rc` is included in the `getstart` subdirectory of the `examples` directory. This file should be copied to your home directory, for example:

```
cp st20rc $HOME/.st20rc                (UNIX and Linux)
copy st20rc %HOME%\st20.rc             (Windows)
```

This start-up file contains the following command:

```
directory /st20tools/sharecfg
```

If your default configuration files are in a directory other than the one given above then you will need to modify the `directory` command accordingly.

2.4.3 Building and running a program

This is exactly the same as before except that this time the tools will find the common file `sti5500.cfg`.

2.5 Creating your own defaults

The toolset is supplied with default definitions of simple simulator targets which can be found in the directory `stdcfg` in the release directory. It is easy to create your own defaults and this example shows how to achieve this.

The filename for any user configuration file must be different to those supplied the `stdcfg` directory. For example, you may create target definitions in a file `mydefs.cfg`. This may be shared by placing it in a shared directory (for example `/st20tools/sharecfg`) or may be kept personal by placing it in a private directory. This configuration file may be included by placing the following line in your start-up file after any relevant `directory` command (see section 2.4.2):

```
include mydef.cfg
```

(The start-up file is called `.st20rc` under UNIX and Linux or `st20.rc` under Windows). Your new defaults will then be seen by the toolset.

2.6 Creating a ROM image

Creating a ROM image is very similar to creating a linked unit file. The difference is that ROM loader code must be linked with the application instead of DCU loader code. This is done by specifying the output format for a ROM image to `st20cc`. In the example below, a ROM image is built, and then run on the simulator and then run on target hardware. This example only works when run via `st20run` as the ROM image produced does not perform the pokes for the External Memory Interface. See Chapter 9 for a fuller description of stand alone ROM systems.

Note: Running this example on EVAL-5500 requires the FLASH memory to be ‘burned’ with the ROM image `hello.hex`. An example flash burn program is provided in the `flash` subdirectory of the `examples` directory.

1 Compile and link for an EVAL-5500 ROM:

```
st20cc hello.c -T sti5500.cfg -p link -g -M hello.map -romimage
```

where:

`hello.c`	: The program to be compiled and linked.
`-T sti5500.cfg`	: Specifies the command file which describes the hardware and application configuration.
`-p link`	: Specifies the command language procedure to execute.
`-romimage`	: Specify that the output file should be a ROM image. By default this is in hexadecimal format.

The command file `sti5500.cfg` specifies the target hardware and the stack and heap requirements of the program.

Note: the ROM file generated is called `hello.hex`, and contains all ROM segments in the target description (in this case there is only one ROM segment).

Open the map file and note that the program code, `def_code`, has been placed with an attribute `MVTORAM`. This indicates that the bootstrap will move the code from ROM to RAM before it is executed.

2 Run the program on the simulator:

ROM files can be executed on a simulator using `st20run`, but not from the command line. Using `st20run` in interactive debugging mode the following can be used:

```
st20run -i sti5500.cfg -t eval5500sim -d
>fill hello.hex
>go
>quit
```

where:

`-i sti5500.cfg`	: Specifies the command file containing the target definitions.
`-t eval5500sim`	: Specifies the simulator target to run the program on.

`-d`	: Specifies interactive mode.
`>fill hello.hex`	: Loads the hex file into memory.
`>go`	: Starts the hex file executing.
`>quit`	: Exits `st20run`.

3 Run the program on the EVAL-5500 target hardware without the debugging GUI:

To run the program from ROM (in which the image `hello.hex` has been burnt) we must connect to the target:

```
st20run -i sti5500.cfg -t eval5500 -d
```

`>go`	: Starts the hex file executing.
`>quit`	: Exits `st20run`.

where:

`-i sti5500.cfg`	: Specifies the command file containing the target definitions.
`-t eval5500`	: Specifies the target to connect and run.
`-d`	: Specifies interactive mode.

4 Debug the program on the EVAL-5500 target hardware with the debugging GUI:

To debug the program in ROM with the debugger we must connect to the target, tell the debugger about the program contained in the ROM and set breakpoints as necessary:

```
st20run -i sti5500.cfg -t eval5500 -g hello.dbg
```

Bring up the Command Console Window (by selecting in the Windows menu) and type:

`>break -h main`	: Sets a breakpoint on main.
`>go`	: Starts the hex file executing.

where:

`-i sti5500.cfg`	: Specifies the command file containing the target definitions.
`-t eval5500`	: Specifies the target to connect and debug.
`-g`	: Specifies graphical interface mode.

See Chapter 9 for a detailed discussion and full example of ROM debugging.

Part 2 - Building and running programs

3 st20cc compile/link tool

3.1 Introduction

This chapter describes the compile/link tool `st20cc`. The chapter gives details of the command line options to run the tool and describes different modes of invoking the tool.

`st20cc` is the driver program to an ANSI C compiler, C++ preprocessor and a compacting linker. `st20cc` provides a simple *single* tool interface which enables you to compile and link your source files to generate:

- a fully developed design in the form of a ROM image suitable for blowing into ROM and running on a ST20 processor;
- a development system in the form of a linked unit, which can then be debugged using `st20run`, and loaded and run on a variety of ST20 targets including a simulator and processor cores;
- a relocatable code unit (RCU) that can be dynamically loaded at runtime.

`st20cc` takes as input user code in the form of C or C++ source, assembly source, object files or libraries, and produces object files, linked units, ROM image files or relocatable code units. `st20cc` is driven by both command line options and a powerful command language which determine the compilation and linkage performed and the type of output file produced.

Figure 3.1 shows `st20cc` in the context of the ST20 Embedded Toolset development model.

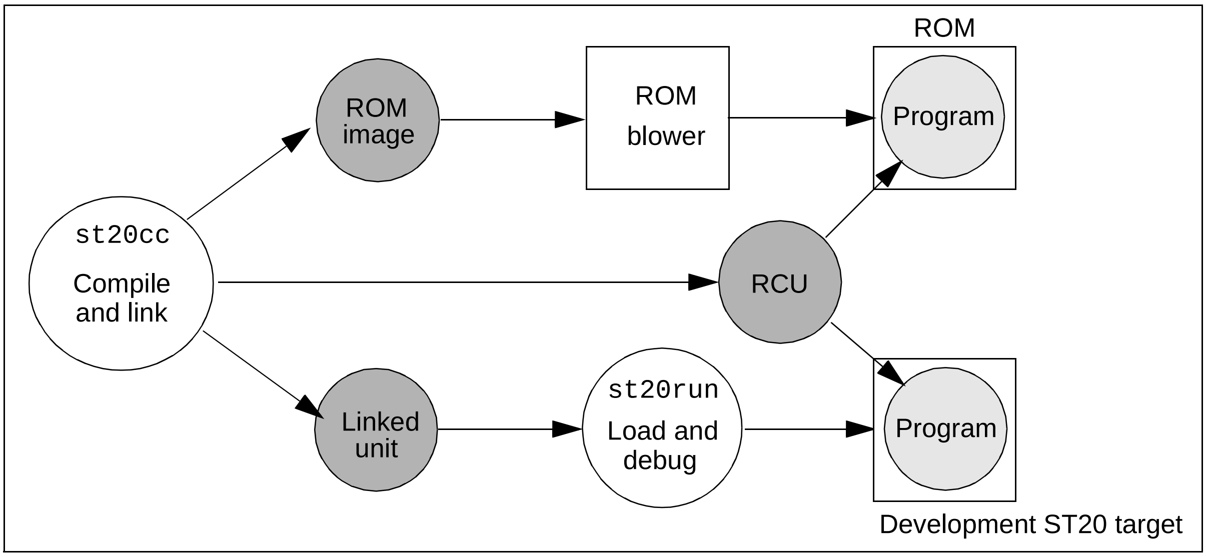


Figure 3.1 `st20cc` development model

Note: information which pertains solely to C++ programs is described separately in Chapter 4. This chapter should still be read by C++ developers, as it is still broadly relevant and contains details of command line options which are common to both C and C++.

3.1.1 Summary of default behavior

By default `st20cc` will:

- Look for and execute a start-up command language script as described in section 1.4.1.
- Preprocess, assemble or compile and then link input files, generating a linked unit in the current directory.
- Compile/assemble as directed by input file suffix, for example, a `.c` file compiles as ANSI C, a `.cxx` file is compiled as C++ and a `.s` file is assembled. See section 3.2.6 and section 4.2.
- Apply local optimizations if applicable and optimize in favor of speed rather than optimum memory usage.
- Append debugging data to assembly files or minimal debugging data to C source files.
- Display any warning messages during compilation.

The following sections describe the options which modify this default behavior of `st20cc`.

3.2 Command line syntax

To invoke `st20cc` use the following command line:

➤ `st20cc` {*options* | *operands*}

where *options* is a list of options given in Table 3.1 and *operands* is a list of input files. Allowable file types are listed in Table 3.2. Filenames may have the following format:

- The first character of a filename may be:
  - alphanumeric
  - an underscore '_'
- Subsequent characters may be:
  - alphanumeric
  - an underscore '_'
  - a commercial at sign '@'
  - a percentage sign '%'

If no input files are specified either directly or indirectly via the command line, and the `-V` option is not specified, then a brief help page is written to the standard error output. A full help page may be displayed by specifying the `-help` option. If no input files are specified via the command line, and the `-V` option is specified, then the version string of `st20cc` is written to standard error output.

If either the `-V` or `-help` options are specified on the command line then all other options are ignored.

3.2.1 Command options

Options may be:

- entered using the environment variable `ST20CCARG`, see section 3.2.9.
- entered directly on the command line,
- entered indirectly via a command file, see section 3.5.4.

If all three methods are used, options will be executed in the above order.

A detailed description of the `st20cc` options is given in Table 3.1. For C++ users, additional options are available which are listed in Table 4.2.

Option	Description
`-A`	Assemble the input file to produce an object file. The compiler phase is suppressed.
`-B` *directory*	Specify the directory containing the tools called by `st20cc`.
`-C`	Run the preprocessor and then terminate. The preprocessed source file is sent to standard output. Compilation is suppressed. Comments are preserved in the preprocessed output.
`-D` *name*`[=`*value*`]`	Define a name. If the *value* is given then a name is defined and assigned that value. This has the same effect as the `#define` preprocessor directive in the C language. See the "*Preprocessing*" chapter in the "*ST20 Embedded Toolset Reference Manual*".
`-EDU`	Force processing of -D (define) before -U (undefine). See the "*Preprocessing*" chapter in the "*ST20 Embedded Toolset Reference Manual*". †
`-H`	Display file searching diagnostics for `#include` preprocessor directives.
`-I` *directory*	Add *directory* to the list of directories to be searched for source files in `#include` preprocessor directives.
`-L` *directory*	Add *directory* to the list of directories to be searched for libraries. This option is passed to the linker.
`-M` *mapfile*	Generate a module information file called *mapfile*. This option is passed to the linker.
`-MA`	Sort symbols in the map file by address.
`-MM`	Sort symbols in the map file by the associated module/library name.
`-MN`	Sort symbols in the map file by symbol name.
`-MS`	Sort symbols in the map file by the defining source file name.
`-N`	Do not copy the debug information from the `.tco` files into the `.dbg` file. See section 3.2.7.
`-NS`	Do not execute any start-up command scripts. See section 3.5.1.
`-O0`	Disable optimization. See section 3.3.4.
`-O1`	Enable local optimization. This is the compiler default. See section 3.3.4.
`-O2`	Enable both global and local optimization. See section 3.3.4.

Table 3.1 `st20cc` options

Option	Description
-P	Run the preprocessor and then terminate. The preprocessed source file is sent to standard output. Compilation is suppressed.
-PPE	Run the preprocessor, generating #line output and then terminate. The preprocessed source file is sent to standard output. Compilation is suppressed.
-S	Compile each source file to assembly language and write it to a file. Assembly is suppressed and no object files are produced. By default output files are placed in the directory from which st20cc is invoked. They are named after the corresponding source input files and given the extension .s.
-T *scriptfile*	Specify a command file. st20cc passes this option on to the linker.
-U *name*	Disable a *name* definition. This is equivalent to the #undef preprocessor directive. See the “*Preprocessing*” chapter in the "*ST20 Embedded Toolset Reference Manual*".
-V	Print version information on standard error as it executes. The default is not to produce this information.
-V18-cmdline-order	Apply options in the order they would have been applied in R1.8 and earlier toolsets. See section 3.2.2.
-W*tool* *arg1, arg2, ...*	Pass the argument(s) as separate arguments to compilation and link tools called by st20cc. Arguments must be separated by commas. *tool* can be one of the following: 0 st20icc ANSI C compiler 1 st20link Toolset linker 3 st20edg C++ preprocessor 6 st20libr Toolset librarian See Chapter 4 for further details of C++ tools. See also, section 3.2.3.
-c	Produce an object file for each source file and suppress the linking phase of the compilation. By default object files are placed in the directory from which st20cc is invoked. They are named after the corresponding source input files and given the extension .tco. If this option is not specified, the default is to perform linking, and to remove all intermediate object files.
-c1	Generate code for the ST20-C1 processor core.
-c2	Generate code for the ST20-C2 processor core.
-cpp	Allow C++ style comments in the source file.
-debug-runtime	Link in the debug run-time kernel. The debug library is used instead of the standard deployment library. This enables access to various runtime kernel debugging features. Further details are provided under the heading "*Debug and deployment kernels*" in section 3.2.5.
-depend [file]	Generate makefile dependencies into the named file. See section 3.7.
-dl	Build an RCU with an entry point that returns import and export parameters, see the "*ST20 Embedded Toolset Reference Manual*".
-e *symbol*	Use *symbol* as the name of the main entry point address of the program. st20cc passes this option to the linker.

Table 3.1 st20cc options

Option	Description
-falign *number*	Change the alignment for nested structs and unions to *number* of bytes. See section 3.3.6 of this manual and the section "*Changing the alignment of structures and unions*" in the "*Implementation details*" chapter of the "*ST20 Embedded Toolset Reference Manual*".
-fcheck-macros	Generate warning messages on #defined but unused macros. †
-fcheck-referenced	Report all externally visible functions and variables which are declared but un-referenced and have file scope. †
-fcheck-side-effects	Provide information on how the compiler has treated routines with respect to side-effects. †
-fdisable-device	Means volatile == access via device instructions. If you do not wish to use this option then the pragma ST_device(*ident*) can be used, see the "*ST20 Embedded Toolset Reference Manual*".
-fdisable-text	Disable checks for invalid text after #else or #endif preprocessor directives. ANSI compliance check. †
-fdisable-type	Disable checks for invalid type casts. ANSI compliance check. †
-fdisable-zero	Disable check for zero sized arrays. ANSI compliance check. †
-filled	When generating a ROM image, cause all unused areas within the ROM to be filled with the value 0xFF. This value may be changed using the -filledval option. This option may not be used with the -rcu option.
-filledval *value*	Cause the byte *value* specified to be used when generating filled ROM images. This is used in conjunction with the filled option. This option may not be used with the -rcu option.
-finl-asm	Inline all functions which include __asm in their definitions. -finl-functions must also be specified.
-finl-ffc	Inline simple function calls that do not precede their call. This option can only be used in conjunction with -finl-functions. See section 3.3.5.
-finl-functions	Inline all simple function calls that do precede their call and that meet the inline function size criteria. See section 3.3.5.
-finl-none	Do not inline-expand functions declared inline.
-finl-timeslice	Insert timeslice instructions for code compiled for ST20-C1 targets.
-finlc *count*	Specify the maximum *count* of call sites for inlining a function. Inlining stops when *count* is reached. Further information is given in section 3.3.5 under the heading " *Command line options for controlling function inlining*".
-finll	Only inline functions which are in loops. Further information is given in section 3.3.5 under the heading " *Command line options for controlling function inlining*".
-finls *size*	Set the maximum function *size* for inlining. Further information is given in section 3.3.5 under the heading " *Command line options for controlling function inlining*".

Table 3.1 st20cc options

Option	Description
`-fp`	Provide floating point support for the functions `atof`, `printf`, `sprintf`, `sscanf`, `strtod` and `vsprintf`. If this option is used ‘`fp`’ will be added to any name supplied to the `-runtime` option, see section 3.2.5.See the “*Libraries introduction*” chapter of the "*ST20 Embedded Toolset Reference Manual*" for more information about the floating point libraries.
`-fshift`	Treat all right shifts of signed integers as arithmetic shifts.
`-fsigned-char`	Change the signedness property of plain `char` and plain bit-fields to be signed. The default is to compile as unsigned.
`-fsoftware`	Perform a number of software quality checks.
`-fspace`	Optimize for space. See section 3.3.4.
`-ftime`	Optimize for time. This is the compiler default. See section 3.3.4.
`-funroll-loops=`*n*	Enable loop unrolling, specifying the maximum number of times to unroll a loop with *n*. *n* should be an integer greater than 1. See section 3.3.4.
`-g`	Generate comprehensive debugging data. The default is to produce minimal debugging data.
`-help`	Display full help information for `st20cc`.
`-in-suffices` *filetype* = *[.ext1, .ext2 ..]*	Allow one or more file extensions to be associated with a file type. Where *filetype* may be one of the following: `cppfile` (C++ files) `cfile` (C files) `linkfile` (Object files) `libfile` (Library files) `asmfile` (Assembler files) See also, section 3.2.3.
`-lib`	Create a library as output.
`-make`	Do a make style date check to avoid recompilation or relinking. See section 3.7.
`-makeinfo`	Display the reasoning behind the make process. This option must be used in conjunction with `-make`, see section 3.7.
`-mpoint-check`	Insert run-time code to check that pointers are correctly aligned, and that NULL pointers are not de-referenced.
`-mstack-check`	Insert run-time code to check that the stack does not overflow.
`-nolibsearch`	Do not search `ST20ROOT/libs` when linking. See section 3.2.4.
`-nostdinc`	Do not compile with standard include paths. See section 3.2.4.
`-nostdlib`	Do not link with standard include files. See section 3.2.4.
`-o` *[outfile]*	Place the output in *outfile*. This applies to an executable file, object file, assembler file or preprocessed source code. For file types other than ROM files, if no filename is given, the tool derives the output filename from the filename stem of the first input object file, adding the appropriate extension. By default the output file is placed in the directory from which `st20cc` is invoked.

Table 3.1 `st20cc` options

Option	Description
`-off` *type*	Set the format of the output file from the linker to *type*; the default is a linked unit (`lku`). Formats other than linked units are all ROM formats and cause a ROM image file to be produced. This option is passed to the linker. See Table 3.3 for possible values of *type* and their meaning. This option may not be used with the `-rcu` option.
`-p` *procedure*	Call the command *procedure*.
`-place-exact`	Force the linker to place sections in the order in which they appear in the configuration (.cfg) file, rather than the order they are defined in the object (.tco) file, see the "*Code and data placement*" chapter of "*ST20 Embedded Toolset Reference Manual*".
`-quiet`	Suppress close down messages from `st20cc`.
`-rcu`	Generate a relocatable code unit (RCU) which may be dynamically loaded. This option may not be specified with any of the ROM generation options that is the `-off`, `-filled` or `-filledval` options.
`-rcuheap` *value*	The value of the heap size for the RCU, expressed in decimal. Defaults to 0. This option may only be specified if the `-rcu` option is used.
`-rculoc` *slot-value*	The value of the user slot in the RCU header. Four fields or '*slot*'s may be defined by the user to hold their own data. *slot* may take a value from 0 to 3 and *value* may be an appropriate value expressed in decimal, up to 8 digits long. Defaults to 0.This option may only be specified if the `-rcu` option is used.
`-rcustack` *value*	The value of the stack size for the RCU, expressed in decimal. Defaults to 0. This option may only be specified if the `-rcu` option is used.
`-romimage`	Create a ROM image output file in hex format. ROM segments not associated with a `romimage` command are output to a single file, see section 3.2.7. An alternative ROM format can be specified using `-off`. `-romimage` cannot be used with either of the `-off lku` or `-rcu` options.
`-runtime` *name*	Selects the runtime libraries to link in.These are used instead of the default runtime libraries. *name* is the name of a configuration file that references the required libraries, see section 3.2.5.
`-search_env {`*path*`}`	Specify a new environment for `ST20ROOT`, which is valid for the current `st20cc` session. Any tools called by `st20cc` will see the new path for `ST20ROOT`.
`-sk` *mapfile*	Generate a map file containing stack depth analysis. See the "*Stack depth and memory maps*" chapter of "*ST20 Embedded Toolset Reference Manual*".
`-suffix [ext]`	Define the extension that object files are named with.
`-t`*tool* *name*	Use *name* as the full pathname of the tool to be invoked by `st20cc`. *tool* can be one of the following: 0 `st20icc` ANSI C compiler - default compiler 1 `st20link` Toolset linker- default linker 3 `st20edg` C++ preprocessor 6 `st20libr` Toolset librarian See Chapter 4 for further details of C++ tools.
`-use-stderr`	Send stdout and stderr to different files. This option is only available on Windows. See section 3.2.10 for further explanation.

Table 3.1 `st20cc` options

Option	Description
`-v`	Display detailed progress information at the terminal as `st20cc` runs.
`-wa`	Suppress messages warning of ‘=’ in conditional expressions. †
`-warn-unused`	`st20cc` warns of any functions or global variables which are not used in the linked program. Potentially these can be removed to reduce memory problems.
`-wc`	Disable nested comment warnings. †
`-wd`	Suppress messages warning of deprecated function declarations. †
`-wf`	Suppress messages warning of implicit declarations of `extern int()`. †
`-wn`	Suppress messages warning of implicit narrowing or lower precision. †
`-ws`	Suppress warning messages about possible side effects. †
`-wtg`	Suppress messages warning of trigraphs. †
`-wv`	Suppress messages warning of non-declaration of void functions. †
`-ww`	Suppress all warning messages.
`-z` *string*	Reserved for internal use.
Entries marked with † after the description are not supported for C++.	

Table 3.1 `st20cc` options

3.2.2 Command line parameter order

The order of parameters on the `st20cc` command line is significant. By default, `st20cc` now passes command line options to sub-tools (for example, the compiler and the linker) in the order that they appear on the `st20cc` command line. There are exceptions to this, for example `-Iincludefile` must come before other include files.

The `st20cc` command line option `-V18-cmdline-order` forces the command line arguments to be passed as they would have been under R1.8 and earlier toolsets. Library files and any directories specified via the `-L` command line option are linked in reverse order. This option must appear first on the command line of `st20cc` and may be made the default behavior by adding the following command to the `st20cc.cfg` file:

```
st20ccoptions "-V18-cmdline-order"
```

3.2.3 Preconfiguring st20cc

In `$ST20ROOT/stdcfg`, there is a command file called `st20cc.cfg` which is used to preconfigure `st20cc`. It is executed when `st20cc` starts up and is written using the toolset command language, see the "*ST20 Embedded Toolset Reference Manual*". Currently this file is supplied with some structured variables which associate file extensions with file types.

Typically these might be set to the following values:

```
## main C compiler
in_suffices.cfile=".c .C .icc .h .H"
## C++ processor
in_suffices.cppfile=".cxx .cpp .CXX .CPP"
## linker objects
in_suffices.linkfile=".TCO .tco .TC1 .tc1 .TC2 .tc2"
## linker libaries
in_suffices.libfile=".LIB .lib"
## assembler
in_suffices.asmfile=".s .S"
```

If the settings supplied in `st20cc.cfg` meet your requirements you may not wish to edit this file. However, you may add to or edit the above definitions and you may also include `commandline` commands in this file.

The `commandline` command, see the "*ST20 Embedded Toolset Reference Manual*", enables you to create a command line option for `st20cc` that triggers an option currently supported by `st20edg`, `st20icc`, `st20link` or `st20lib`. In effect `commandline` enables you to abbreviate the `-W`*n* `st20cc` command line option. The `commandline` command should only ever be executed from `st20cc.cfg`, as this is seen only by `st20cc`.

Note: the `st20cc` command line options, `-in-suffices` and `-W`*n* can still be used. `-in-suffices` specified on the command line may override settings defined in `st20cc.cfg`.

3.2.4 File search rules

All tools in the toolset, including `st20cc`, access system files via the environment variable `ST20ROOT`, which contains the root directory pathname for the toolset. If the tools called by `st20cc` are in a different directory, the directory pathname can be supplied on the command line by using the `-B` option.

`st20cc` also recognizes the `directory` command language command, as described in the "*ST20 Embedded Toolset Reference Manual*". In addition the `-I` and `-L` options to `st20cc` enable you to specify additional directories for `st20cc` to search for source files when compiling and linking respectively. Any directory specified by the `-I` option will be searched first before the root directory defined by `ST20ROOT`.

Directories are searched in the following order:

- For files included in the compilation by the `#include <....>` predefine:
  - i `-I` directories specified on the command line
  - ii `$ST20ROOT/include` (header files supplied with the toolset)
- For files included in the compilation by the `#include “....”` predefine:
  - i current directory
  - ii `-I` directories specified on the command line
  - iii `$ST20ROOT/include` (header files supplied with the toolset)

- For files specified to `st20cc` for linking:
  - i current directory
  - ii `$ST20ROOT/lib` (libraries supplied with the toolset)
  - iii `$ST20ROOT/stdcfg` (default definitions supplied with the toolset)
  - iv `-L` directories specified on the command line
  - v directories specified with the `directory` command in command files, see section 3.5.3.

The command option `-search_env` {*path*} enables a new environment to be set for `ST20ROOT`, for the current session of `st20cc`.

The command options `-nostdlib` `-nostdinc` and `-nolibsearch` can be used to stop default searching:

- `-nostdlib` instructs `st20cc` not to link in the C libraries. If `-nostdlib` is used, a main entry point *must* be specified on the command line by using the -e option. Any libraries required must be specified either directly on the command line, see section 3.2.6, or via a configuration file using `file` commands to reference the library files, see section 3.5.
- `-nolibsearch` instructs `st20cc` not to use any of the files in `$ST20ROOT/libs`. This option would be used in conjunction with the `-L` option to specify and access the user's own libraries. If `-nolibsearch` is used, a main entry point *may* need to be specified on the command line by using the -e option.
- `-nostdinc` instructs `st20cc` not to use any of the files in `$ST20ROOT/include`. This option would be used in conjunction with the `-I` option to specify and access the user's own header files.

File searching diagnostics can be displayed by specifying the `-H` option, this will display a message for each `#include` preprocessor directive used.

3.2.5 Runtime libraries

The runtime libraries can be selected by specifying the `-runtime` *name* command line option. *name* is the name of a configuration file which uses `file` commands to reference the required library files.

The configuration filename takes the form:

nametype[*fp*]`.cfg`

Where: *name* can be:

- `c` - selects the default runtime libraries;
- `os20` - selects the STLite/OS20 runtime libraries;
- `psos` - selects the pSOS runtime libraries;
- *name* - a name which selects a user defined runtime library.

  name is supplied as an argument to the `-runtime` option.

type is either `lku`, `rom`, or `rcu` depending on whether the runtime library supports a linked unit, a ROM image file or a relocatable code unit. *type* must be included in the filename but is not specified to `-runtime`.

fp is optional and selects a runtime system which uses the floating point mathematics libraries. If the `-fp` command line option is used ‘`fp`’ will be automatically added.

`.cfg` denotes a configuration file, see section 3.5.

For example:

```
myruntimelku.cfg
myruntimerom.cfg
muruntimelkufp.cfg
```

`st20cc` will select the configuration file depending on the name supplied to `-runtime` and on whether it is creating a linked unit or ROM image file.

For example:

```
st20cc hello.c -runtime myruntime -T appcontrol.cfg -p helloapp
```

In this example `st20cc` creates a linked unit, so the runtime libraries are selected from the file `myruntimelku.cfg`.

Debug and deployment kernels

When selecting the runtime libraries, `st20cc` also supports the selection of either a ‘*deployment*’ kernel or a ‘*debug*’ kernel. By default `st20cc` uses the deployment kernel associated with the selected runtime library. At present the only runtime libraries which are supplied with a pre-built debug kernel are the STLite/OS20 runtime libraries.

A debug kernel differs from a deployment kernel in that it has additional debugging facilities. These facilities will vary between runtime libraries, for example, the STLite/OS20 debug kernel includes a time logging capability, which can help the developer analyze certain characteristics of their system or isolate problems. Once development is complete and the need for debugging features has ceased the user may switch over to using the deployment kernel; typically the deployment kernel is the kernel which is included in the final delivery. The deployment kernel does not include time-logging.

The debug kernel is selected by using the `-debug-runtime` option to `st20cc`, which selects the prebuilt debug kernel library rather than the standard deployment kernel library. For example, the following command line will create a linked unit using the debug version of the STLite/OS20 run-time kernel:

```
st20cc hello.c -debug-runtime -runtime os20 -T sti5500.cfg -p link
```

Once debugging is complete, the application can be rebuilt using the deployment kernel by just removing the `-debug-runtime` option from the command line:

```
st20cc hello.c -runtime os20 -T sti5500.cfg -p link
```

For further details of the STLite/OS20 runtime kernel see Chapter 12.

3.2.6 Input files

Table 3.2 lists the accepted input file types to `st20cc`. The file extensions listed in Table 3.2 should be used so that `st20cc` can recognize the file type, they may be in upper or lower case.

File type	Description
`file.c`	A C language source file which is to be preprocessed, compiled and linked.
`file.h`	A C language header file which is to be passed to the compiler.
`file.s`	An assembly language source file which is to be assembled and linked.
`file.tco`	An object file to be passed directly to the linker.
`file.lib`	A library file to be passed directly to the linker.
`file.cfg`	A command language file, used to control linking, which is passed directly to the linker.
Note: C++ input file types are listed in section 4.2	

Table 3.2 `st20cc` input file types

Command files are specified on the command line using the `-T` option. All other input file types may be specified directly on the command line.

C source files are by convention given a `.c` filename extension. If a source file is specified without an extension or with a non-standard filename extension, `st20cc` will assume it is a C source file.

Assembler source files are by convention given a `.s` extension and must be given an extension other than `.c`.

UNIX/Windows incompatibility

Files in MS-DOS/Windows format should be converted to UNIX format to be compiled under UNIX. **Note:** Windows is insensitive to the case of filenames (for example `A.c`, `a.c`, `a.C` and `A.C` are all the same). This can have an effect on ROM image files and hence memory segments.

3.2.7 Output files

`st20cc` can be invoked to compile, link, assemble or preprocess input. Figure 3.2. illustrates the main output file types which may be generated.

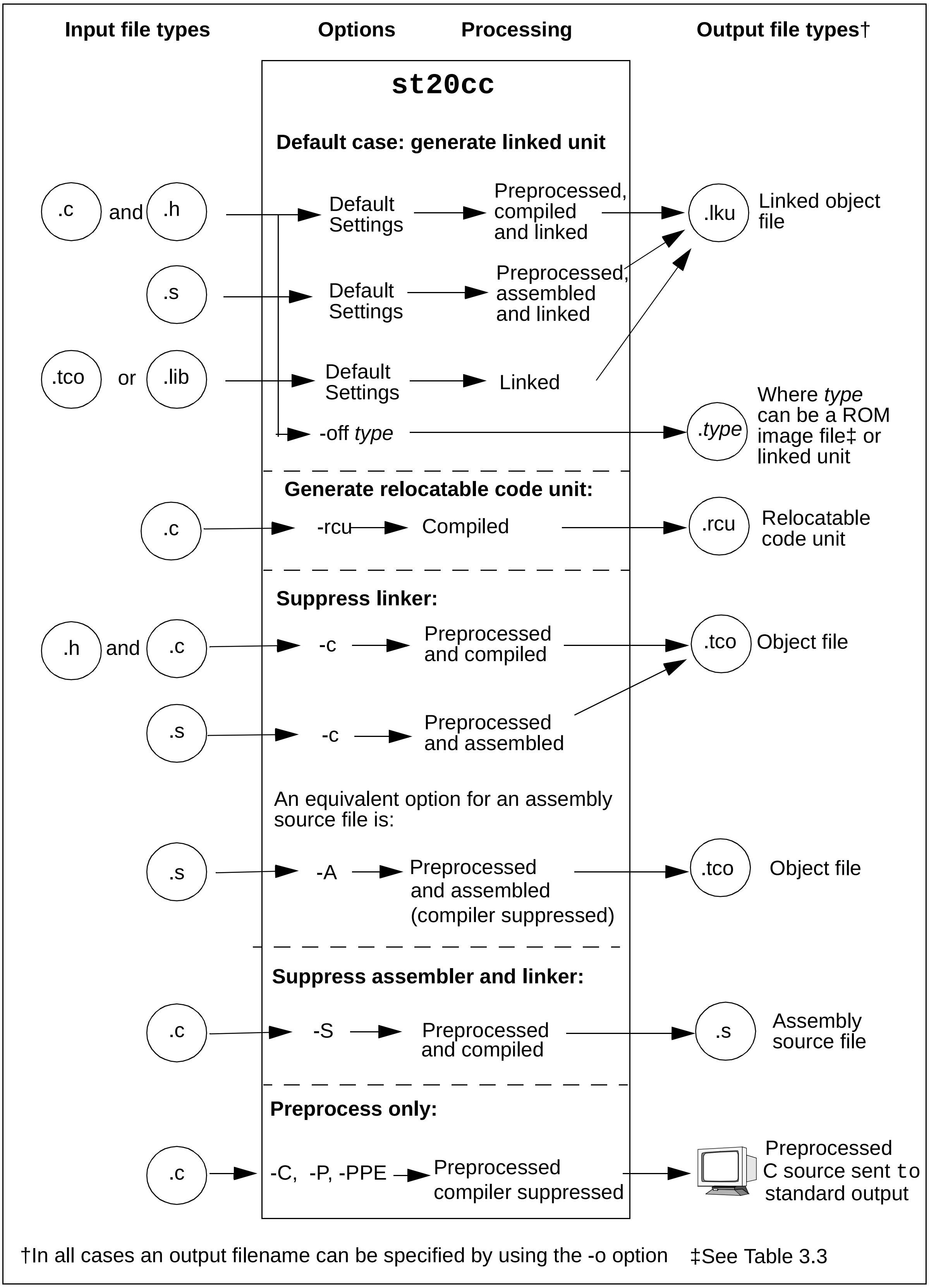


Figure 3.2 `st20cc` primary output files

By default `st20cc` will attempt to build a linked object file also known as a linked unit. C source files and header files will be preprocessed, compiled and linked. Assembly source files will be preprocessed, assembled and linked. Input files which have already been compiled (that is library files and object files) will be linked.

The following options are used to suppress the linker, compiler or assembler:

- Option `-c` suppresses the linker. The input will be preprocessed and either assembled or compiled to produce object files.
- Option `-S` suppresses the linker and assembler and will generate preprocessed assembly source files.
- Options `-C`, `-P` and `-PPE` invoke the preprocessor and suppress the compiler. Output will be directed to standard output unless an output file is specified.
- Option -A can be used to process an assembly source file and is equivalent (in this case) to option -c.
- Option `-rcu` can be used to generate a relocatable code unit, which may then be dynamically loaded. This option must not be used in conjunction with options to generate a ROM image file.
- Option `-lib` can be used to generate a library as output.

For file types other than a ROM image file, the tool derives the output filename from the filename stem of the first input object file or the name specified using the `-o` option and adds the correct extension:

- `.`*ext* for a ROM image file, as shown in Table 3.3;
- `.lku` for a linked unit file;
- `.tco` for a compiled object file;
- `.s` for an assembly source file;
- `.dbg` for a debug information file;
- `.rcu` for a relocatable code unit;
- `.lib` for an object code library.

Compiled object files may have the default extension `.tco` changed by specifying the `-suffix` command line option.

A debug information file is always generated in addition to the linked unit or ROM image file. If the `-N` option is used the debug information in the `.tco` files is not copied into the `.dbg` file. In order for the debugger to find the debug information, either the `.dbg` or the `.tco` files must be in the same directory as the associated `.lku` file. If, for any reason, the `.lku` file is copied or moved to a different directory then the `.dbg` file should be moved or copied to the same directory. If the `-N` option is used it is good practice to keep all the `.tco` files in a single build tree. If the debugger cannot find the `.dbg` file, in interactive mode it prompts the user for a pathname.

When producing a linked unit file, `st20cc` can be instructed to generate a memory map and a map of stack depth analysis, see section 3.2.8.

Specifying an output file using -o

The `-o` option enables an output file to be specified and can be used with any of the file types generated by `st20cc`. Filenames may only contain alphanumeric characters or underscores '`_`' . The filename specified will be used for the output filename (unless it is a ROM image, see below) and to derive the name of the debug information file.

If more than one `.c` or `.s` file is specified on the command line then the `-o` option should only be specified if the command generates a single linked output file, either a linked unit or a ROM image file. If linkage is suppressed by using any of the `-c`, `-S` or `-P` options then multiple input files will cause multiple output files to be generated, thus conflicting with the use of the `-o` option. In this case if the `-o` option is specified, an error occurs and no processing takes place.

Specifying a ROM image file

The `-romimage` option can be used to specify a ROM image file in hexadecimal format. If the `-romimage` option is supplied then any ROM segments not associated with a `romimage` command are output to one ROM output file. The format of this file is derived from the `-off` option (see below), and if that is not supplied then defaulted to type `hex`. The name of the ROM image output file is supplied by the `-o` option, or derived from the first input object file with a filename extension dependent upon the ROM type. See also section 3.5.5.

Alternatively, the `-off` *type* option can be used to specify a ROM file in a different format, where *type* may be one of the values listed in Table 3.3. The output file is placed in the directory in which `st20cc` is invoked. The name of the output file is derived from the associated ROM segment name and the extension from the ROM image type. If an alternative filename is required for the ROM file then the `romimage` command should be used, see section 3.5.6.

`-off` *type*	**Filename extension**	**Description**
`binary`	`.bin`	Binary ROM format
`hex`	`.hex`	Hexadecimal ROM format
`lku`	`.lku`	Linked unit
`srecord`	`.mot`	Motorola S-Record ROM format

Table 3.3 Input and output formats to `-off` *type*

The default output format for `-off` is `.lku`, that is a standard linked unit.

ROM image files are compacted by the linker in order to reduce the size of ROM required to store the application code.

3.2.8 Map files

`st20cc` can be instructed to generate a memory map file which gives detailed information about:

- the size of the program;
- how much space there is remaining for data;
- which libraries are linked in;
- where symbols have been placed in memory.

Map files are of particular use when placing code and data, (see section 3.4), to see exactly where symbols have been placed. Map files are generated by specifying the `-M` option on the command line.

The `-sk` command line option can also be used to give an analysis of the stack requirements of the program and is of particular use when space is limited and the stack is required to be kept to a minimum.

Both types of map file are described in detail in the “*Stack depth and memory maps*” chapter of "*ST20 Embedded Toolset Reference Manual*".

3.2.9 Environment variable

`st20cc` options and operands may be specified using the environment variable `ST20CCARG`. This provides an easy method of setting up default values for `st20cc` command line options. Options and operands specified by this method will be treated as though they appeared on the command line.

The syntax for `ST20CCARG` is:

[*options*]

Options specified via `ST20CCARG` are placed before the command line options to `st20cc`. As later options override earlier ones, options specified via the command line may override those specified by `ST20CCARG`. If `ST20CCARG` is not present then it is treated as if it contained no options.

3.2.10 Errors

If a compilation or assembly error occurs for an operand, `st20cc` will continue to compile or assemble subsequent operands; however, the link and bootable generation phase will not be performed.

If the `-o` option and more than one `.c` or `.s` file is specified then an error is flagged and no compilation, assembly or preprocessing takes place.

Standard error stream on Windows

On Windows hosts it may be useful to send stdout and stderr to different files and this can be done using the option `-use-stderr`. This option is not available in UNIX or Linux. For example, to send all output to file `std.out`:

```
st20cc -p c2 demo.c > std.out
```

The following NT command line separates stdout and stderr and redirects them to two files `std.out` and `std.err` respectively:

```
st20cc -p c2 -use-stderr demo.c > std.out 2> std.err
```

3.2.11 Example st20cc command lines

```
st20cc hello.c -T appcontrol.cfg -p helloapp
```

This example compiles and links the source file `hello.c` into a linked unit file named `hello.lku`. The name of the output file is derived from the input file and the extension `.lku` added. `appcontrol.cfg` is a command file which contains a command language procedure called `helloapp` which is specified to describe the target device and to control linking, see section 3.5 for further details. The example is compiled for an ST20-C1 processor core which is defined by the command procedure. A debug information file `hello.dbg` will also be generated.

```
st20cc hello.c -off hex -T appcontrol.cfg -p helloapp
```

This example is similar to the previous example, except that instead of generating a linked unit, the command outputs a hexadecimal ROM image file.

```
st20cc -c1 -c hello.c
```

This example compiles the source file `hello.c` into an object file named `hello.tco`. The name of the output file is derived from the input file and the extension `.tco` added. The command compiles for a ST20-C1 processor core and linkage is suppressed. See section 3.3.1.

3.3 Compilation

This section describes the compilation facilities and options provided by `st20cc`. The toolset compiler used by `st20cc` is an ANSI standard C compiler which conforms to *ISO/IEC 9899:1990 Programming languages - C*. When processing C++ code, `st20cc` invokes the Edison Design Group preprocessor to convert C++ into C, see Chapter 4 for details.

3.3.1 Selecting the processor core

A target processor core must be selected; there is no default. The processor cores supported are the ST20-C1 and ST20-C2. These may selected by specifying the `-c1` and `-c2` options respectively. The recommended method is to specify either the `chip` or `processor` command within a procedure in a startup file, and call the procedure on the `st20cc` command line using the `-p` option. See section 3.5.1 and section 3.5.5.

If a `chip` or `processor` command contradicts the use of the `-c1` and `-c2` `st20cc` command line options, then `st20cc` will generate an error.

3.3.2 Using the assembler

Assembler source files which have the `.s` file extension will be automatically passed to the assembler, suppressing the compilation phase. Any preprocessor directives in the input assembly source file will be preprocessed.

Assembly files have basic debug data generated automatically and can be debugged by `st20run`. A number of macros are provided in `asmdebug.h` which may be used in assembler source files to allow backtracing and symbolic debugging. These are documented in the “*Support for assembly debugging*” chapter of the "*ST20 Embedded Toolset Reference Manual*".

Note: the `-S` option will compile a C source file into an assembler source file. The assembler phase is suppressed.

The use of the assembler is described in the “*Assembly-level programming*” chapter of the "*ST20 Embedded Toolset Reference Manual*", together with examples of how it is invoked. The file name conventions for assembler files and the command options which may be used with the assembler are listed there. That chapter also describes the syntax of assembler directives.

3.3.3 Using the preprocessor

The preprocessor is automatically invoked as part of assembly and compilation for all C source or assembly files. In addition source and header files may be submitted to `st20cc` for ‘preprocessing only’ by using the `-C`, `-P or -PPE` command line options. The preprocessor implements translation phases 1 to 4 of the ANSI Standard, section 2.1.1.2. The `-C` option additionally preserves comments in the preprocessed output and the `-PPE` option outputs `#line` information. These three options suppress compilation and can be thought of as ‘preprocess-only’ mode.

The preprocessor is used to resolve preprocessor directives (that is directives with a ‘`#`’ prefix) and to perform macro expansion. **Note:** that the preprocessor both resolves any `#pragma` directives it encounters in the input and preserves them in the preprocessed output, in order that they may be re-processed by other tools. The output from preprocess-only mode may be fed back into the compiler or assembler for further processing.

In preprocess-only mode, a target processor type need not be specified. In this case the preprocessor symbol `__CORE__` takes the value ‘`-1`’ which is a dummy value. If a target processor is specified then `__CORE__` will be set as normal.

Preprocess-only mode generates a text file which by default is sent to standard out (`stdout`). The `-o` command line option can be used to direct this output to a file.

The ‘*Preprocessing*’ chapter of the "*ST20 Embedded Toolset Reference Manual*" describes the preprocessing directives and macros supported by `st20cc`.

3.3.4 Compiling with optimization enabled

Compiling source code with optimization enabled generates highly efficient object code. The purpose of optimization is to improve the execution time of object code as well as the program's use of memory, that is workspace, stack and code size. The options `-ftime` and `-fspace` control whether the optimization performed is predominantly to improve execution time or memory use. Optimization does not affect the functionality of the program. Compile times will be slower when a high level of optimization is performed. `st20cc` implements both local and global levels of optimization:

- The global optimizations include: constant propagation, common subexpression elimination, strength reduction, loop invariant code motion and tail-call optimization. The optimizer examines each function as a single unit, enabling it to obtain as much information as possible about that function, while performing the optimization. Global optimizations are more complex than local optimizations; generally the more information available to the optimizer the better chance the optimizer has of improving code. The pragma `ST_nosideeffects` enables the behavior of individual functions to be clarified. The `#pragma` preprocessor directive is described in the "*ST20 Embedded Toolset Reference Manual*".
- The local optimizations include: flowgraph, peephole and redundant store elimination. To perform these optimizations efficiently, the optimizer only needs to operate on short sequences of code.

When optimization is enabled `st20cc` is also able to perform inline function expansion which also improves the execution time of the program by removing function calls, see section 3.3.5.

Advantages of enabling optimization

The advantages of enabling the optimizing features of `st20cc` are that:

- It saves development time by relieving the need to optimize code manually.
- Because the compiler can be relied upon to transform code into a more efficient form, more emphasis can be placed on writing more readable, and hence more maintainable code.
- There are some optimizations which cannot be written into the source code and can only be performed by the optimizing compiler at compile time.
- The compiler is able to analyze the cost against effectiveness of potential optimizations and will only apply an optimization where a saving can be made in either execution time or memory space.

When optimization should not be used

If it is required to debug a program, it may be wise to disable optimization using the `-O0` command line option. `st20cc` will generate debug information when optimization is enabled if requested with the `-g` option. However, the debugger will produce more accurate results if optimization is disabled at compile time.

Also note that assembler code inserted with the `__asm` statement is not globally optimized by `st20cc`. Global optimization is suppressed for any function which contains the `__asm` code and a warning message is generated. The `__optasm` statement should be used instead, see the “*Assembly-level programming*” chapter of the "*ST20 Embedded Toolset Reference Manual*".

Optimization options

- Disable optimization `-O0`

  The option `-O0` disables all optimization which can be specifically enabled at the command line using the `-O1` and `-O2` options.

- Enable local optimization `-O1`

  This option is enabled by default and applies the following local optimizations:

  - Flowgraph optimization, including dead code elimination.
  - Peephole optimization.
  - Redundant store elimination.

  The global optimization workspace allocation by coloring is also enabled.

- Enable local and global optimization `-O2`

  This option, enables the following local and global optimizations:

  - All optimizations enabled by option `-O1`.
  - Constant propagation.
  - Global common subexpression elimination.
  - Loop invariant code motion.
  - Strength reduction.
  - Tail-call optimization.
  - Tail recursion optimization.

- Optimize for time `-ftime`

  This option controls how optimization is applied once it has been enabled by either the `-O1` or `-O2` options. It instructs `st20cc` to perform only those optimizations which will not reduce the speed of the program. Where a choice exists between generating faster, but larger code over slower, more compact code, it will generate the faster code. This option is enabled by default.

- Optimize for space `-fspace`

  The reverse of the `-ftime` option, that is, only performing those optimizations which will not increase the size of the program. Where a choice exists between generating faster, but larger code over slower, more compact code, it will generate the more compact code.

- Optimize with loop unrolling

  The `-funroll-loops=`*n* option may be used in conjunction with other optimizing options. Specifying various integers greater than `1` for value will generally, but not always, lead to a significant performance increase. Almost certainly specifying this option will result in a slight increase in code size.

There is no definitive list of which optimizations will be applied for each of the `-ftime` or `-fspace` options; this will vary depending on the code being optimized. There are circumstances where code optimized with `-fspace` will run faster than with `-ftime`. On critical sections of code it may be worth seeing the effect of this optimization.

Enable side effects information messages

The `-fcheck-side-effects` option enables the generation of information messages about the ‘side effect’ characteristics of functions as `st20cc` performs optimization. The messages report the actions of `st20cc` to give visibility of how functions are treated with respect to side effects. The messages are purely informational and do not signal any required user response.

Side effects are discussed as part of the description of the pragma `ST_nosideeffects` in the "*ST20 Embedded Toolset Reference Manual*".

Disable side effect warning messages

The `-ws` option disables messages warning that functions marked as side effect free may in fact still cause side effects.

Language considerations

Before `st20cc` can optimize a function call it has to be sure that the optimization will not effect the functionality of the code. Therefore it will treat function calls with caution, assuming that they may modify global variables, unless it can deduce with certainty their true behavior.

The following language features affect the implementation of optimization.

- `const` keyword

  The `const` keyword states that after it is initialized, a variable cannot subsequently be modified by the program. For a `const` variable, `st20cc` does not have to make worst case assumptions about its being modified when ambiguous modifications are seen. If a variable is never modified, then declaring it as `const` will, in general, allow `st20cc` to do a better job of optimizing. **Note:** when pointers to `const` objects are used, for example:

  ```
  const char *p;
  ```

  the `const` keyword does not guarantee that the `char` will not be modified, just that it will not be modified through pointer p.

- `volatile` keyword

  The `volatile` keyword states that a variable may change asynchronously, or have other unknown side effects. `st20cc` will not move or remove any loads or stores to a volatile variable. `volatile` should be used for variables shared between parallel threads (or variables modified by interrupt routines), or variables which are mapped onto hardware devices.

- `register` keyword

  The `register` keyword is taken as a hint to the compiler to allocate the variable at a small workspace offset.

3.3.5 Inline code

The toolset provides different levels of support for inlining instructions and functions:

- Special keywords `__asm` or `__optasm` can be used to insert sequences of assembly level instructions into C programs. These keywords and their use are described in the “*Assembly-level programming*” chapter of "*ST20 Embedded Toolset Reference Manual*".
- A facility known as ‘*inline function expansion*’ enables a function body to be substituted in place of a call to the function. It is supported as follows:
  - The keyword `__inline` can be used in a function definition or declaration to specify that the function is to be expanded inline.
  - The pragma `#pragma ST_inline` can be used after the function declaration to specify that the function is to be expanded inline.
  - A command line option is provided to enable *automatic* inlining of simple functions.
  - Further command line options are provided to control the extent of inline function expansion applied by the compiler.

Inline function expansion

Inline function expansion is the substitution of the function body in place of a call to the function.

Inlining functions may improve the execution time of the program, because the inline substitution of the function body removes the overhead of a function call and also creates the potential for further optimizations such as constant propagation. Local and global optimizations selected by the command line options and described in section 3.3.4, are performed by `st20cc` after it has performed any inline function expansion. The improvements in program efficiency must be balanced against the other effect of inlining which is, that the code size of the program’s executable is potentially larger.

You may explicitly mark a function as a candidate for inlining or may use a command line option to specify that `st20cc` is to consider all simple functions as candidates for inlining. In either case the compiler will only inline a function provided various criteria are met, which are described below.

Criteria for inlining

`st20cc` will attempt to apply inline function expansion provided the following criteria are met:

- Optimization is enabled - this is the default. Optimization is only disabled if the command line option `-O0` is specified.
- The compilation generates only minimal debugging information, which is the default. The command line option -g which generates full debugging information suppresses inline function expansion.
- The function definition and the function calls are in the same compilation file.
- The function definition precedes the function call, *or*

  the function definition does not precede the function call and the command line option `-finl-ffc` is specified. (See below.)
- The function has a constant number of formal parameters.
- The function is called directly and not via a pointer.

When the function is recursive then the first call to the function will be expanded inline, subsequent recursive calls to the function, will not be expanded.

In-line assembly code

Functions which contain `__asm` code may be in-lined, however global optimization is disabled for such functions. If the `__optasm` statement is used in place of any `__asm` statement, then the compiler will be able to apply global optimization to the assembly code, including inline function expansion.

Note: the use of `__optasm` does carry some restrictions. Both assembly constructs are described in the "*ST20 Embedded Toolset Reference Manual*".

Reducing code size

For those applications where code size is critical, it may be useful to consider the following optimization which `st20cc` implements:

If `st20cc` can reduce the code size of the object file by *not* generating redundant code then it will do so. For example, when inline function expansion is performed the body of a function will be made redundant if all the following conditions are met:

- For all calls to the function, inline function expansion is performed.
- The address of the function is not referenced, that is, it is not accessed through a pointer.
- The function is not externally visible or shared by other files.

If the above conditions are met then `st20cc` will not generate code for the function.

Marking a function for inline expansion

The two mechanisms provided to enable you to indicate that a specific function is to be compiled inline are:

- The keyword `__inline` which can be used in a function declaration or function definition, as in the following example:

```
static __inline int fn_add(int x, int y)
{
  return X + Y;
}
```

- The pragma `#pragma ST_inline` which can be used after the function declaration, but before function definition:

```
int fn_add(int x, int y);
#pragma ST_inline (fn_add)
```

A call to the function `fn_add` (as defined by one of the above mechanisms), for example:

```
a = fn_add(b, c);
```

would cause the compiler to generate the following code:

```
a = b + c;
```

These two mechanisms are equivalent to each other, and it is a matter of programming style as to which is adopted.

Selecting automatic inlining

When the `-finl-functions` option is specified, *all* simple functions in the compilation file, not just those marked with the `__inline` keyword or the `ST_inline` pragma, are treated as potential candidates for inlining.

The `-finl-functions` option instructs `st20cc` to automatically inline simple function calls provided the following condition is met, in addition to those already listed above:

- the code size of the function is no more than 20 instructions in the intermediate compiler language representation, used by `st20cc`. In the intermediate representation, a sequence of instructions has at most one operator and a single intermediate instruction equates to between 1 and 3 assembly level instructions. For example, a direct load or store intermediate instruction maps to a single assembly instruction, however, instructions which include an operator such as addition equate to three assembly instructions.

If, your code also contains `__asm` code which you wish to be automatically in-lined then you must specify the `-finl-asm` option in addition to the `-finl-functions` option. Global optimization is suppressed.

Command line options for controlling function inlining

A number of command line options are provided to control the inlining of functions within a file. These options apply to functions marked for inlining by the `__inline` keyword or the `ST_inline` pragma as well as to functions affected by the use of the `-finl-functions` option.

- The `-finl-ffc` option instructs `st20cc` to inline function calls which precede their definition. This option is potentially expensive in terms of memory used and compilation time. It should therefore be used with caution. The `-finl-ffc` option must be used in conjunction with the `-finl-functions` option, and removes the restriction that the function definition precedes the function call.
- The `-finl-none` option instructs `st20cc` to *not* inline-expand functions declared inline.

The following three options provide some control over the size of the final object file. These options apply to both explicit inlining with the `__inline` keyword or the `ST_inline` pragma and to automatic inlining selected with the `-finl-functions` command line option.

- The `-finls` *size* option allows the maximum code size of a function, for which `st20cc` may apply inline function expansion, to be specified. *size* is measured in intermediate instruction representation where a single intermediate instruction approximately equates to between `1` and `3` assembly level instructions.
- The `-finll` option instructs `st20cc` to inline functions only in loops.
- The `-finlc` *count* option allows the maximum number of call sites to be specified for inline function expansion.This option stops inlining when *count* is reached. For example, if there are ten nested function calls and `-finlc 2` is used with `-finl-functions`, the first two calls are inlined, the remainder are not.

3.3.6 Compatibility with other C implementations

A number of options are provided which may assist with the porting of existing C code to the ST20 family of processors.

Arithmetic right shifts

By default, `st20cc` implements right shifts of signed integers as logical shifts; the command line option `-fshift` switches the implementation. This allows correct working of programs which assume that right shifts of signed values propagate the sign.

Signedness of char

By default `st20cc` implements plain `chars` as `unsigned chars`. The command line option `-fsigned-char` switches the implementation to `signed char`, plain bit-fields are also signed. Details of type representation are given in the “*Implementation details*” chapter of the "*ST20 Embedded Toolset Reference Manual*".

Alignment of structs/unions

By default, `st20cc` aligns `structs` and `unions` on a word boundary. The command line option `-falign` *number* modifies any nested structures or unions to the specified byte boundary. The `-falign` option does not affect the packing of the structure members themselves. Thus, the alignment will never be less than that required for any element of the `struct` or `union`. The use of this option is described in more detail under the heading "*Changing the alignment of structures and unions*" in the "*Implementation details*" chapter of the "*ST20 Embedded Toolset Reference Manual*".

Some C implementations align a `struct/union` according to the strictest alignment requirements of the fields of the `struct/union`; this can be achieved using the `-falign1` option.

3.3.7 Software quality check

The `-fsoftware` option allows policing of software quality requirements. The option requires all externally visible definitions to be preceded by a declaration (from a header file), thus guaranteeing consistency.

When the `-fsoftware` option is used `st20cc` reports:

- all forward **`static`** declarations which are unused when the function is defined.
- all repeated macro definitions (this is when macros are redefined to the same value; redefining a macro to a different value is always diagnosed as an error).

3.3.8 Runtime checking options

The `-mpoint-check` and `-mstack-check` command line options cause the compiler to insert run-time code to perform checking.

Details of how to generate and interpret an analysis of stack depth are given in the "*Stack depth and memory maps*" chapter of the "*ST20 Embedded Toolset Reference Manual*".

Enable pointer dereference checks

When the `-mpoint-check` option is specified, `st20cc` inserts a check each time a pointer is dereferenced. This check ensures that the pointer is not `NULL` and that the pointer is correctly aligned for the type of object being accessed. For example, in the following code,

```
int *pi;
char *pc;
*pc = (char)(*pi);
```

two pointer checks will be inserted, one to check that `pi` is not `NULL` and that it points to a word-aligned object, and another to check that `pc` is not `NULL`.

Note: that no check is inserted if a pointer is assigned or read but not dereferenced. For example, no checks will be inserted in the following code:

```
int *pi1, *pi2;
pi1 = pi2;
```

Enable stack checks

When the `-mstack-check` option is specified, `st20cc` inserts a check on entry to each function. This check ensures that the stack has enough space available for the function's workspace, plus a margin for calling library functions which do not contain stack checks. This margin is currently 150 words.

As the stack check always ensures that there is a margin free below a function's workspace, any leaf functions (functions which do not call other functions) whose workspace fits into this margin do not require a stack check. `st20cc` will automatically suppress the stack check for such functions.

The function that is called to perform the stack checking is dependant on the runtime system used. The default version will only work with single threaded programs. For information which is relevant when STLite/OS20 is used, see section 14.13.

3.4 Code and data placement

The ST20 toolset provides users with fine granularity code and data placement. The user is given control over the placement of all symbols in memory. The mechanism for doing this operates at two levels:

A program is made up of sections such as `code`, `const`, `data`, `bss`. The user can define their own sections. If they do not define any sections then a set of default sections are used and `st20cc` generates a linked list of sections with default ordering.

The user is also able via the toolset command language to define a memory map for the application program in detail, specifying start and end addresses and the memory type for specific ranges of memory. Named sections from the application program can then be placed into named memory segments.

By defining a memory segment to start at a particular address, it is possible to force a symbol to be placed at a particular address.

Code and data placement is described in the "*ST20 Embedded Toolset Reference Manual*". See section 3.5.7 of this chapter for a list of commands.

3.5 Command language

`st20cc` can interpret the command language introduced in Chapter 1 and defined in the "*ST20 Embedded Toolset Reference Manual*". `st20cc` uses a number of commands to control the linking of the application and optionally, the format of any ROM file.

The command language subset supported by `st20cc` enables you to build a recipe for linking your application and also provides a record of how the linkage was performed.

Commands are submitted to `st20cc` via one or more command files which may be entered on the command line using the `-T` option, for example:

```
st20cc hello.c -T hello.cfg -p phone
```

Where: `hello.c` is the application source file,

`-T hello.cfg` specifies a command file

and `-p phone` calls the command procedure `phone` which is specified in `hello.cfg`.

Commands may also be specified in default command files, see section 3.5.1.

The commands which are used to describe the application and load it onto a target are known as configuration commands. `st20cc` and `st20run` both use configuration commands. `st20cc` uses configuration commands to define the target and describe its memory, to specify how code and data is to be placed in memory and to describe the format of ROM images. `st20run` uses configuration commands to load the code onto the target. `st20run` also uses the command language interactively to debug the running code. Because the tools use a common language and share the use of certain commands, for example the configuration commands which define the target and memory, there are opportunities for sharing common command definitions and procedures. For this reason some thought should be given as to how command files will be used. For single-user, single target systems, all configuration commands may be placed into a single file. For larger systems with multiple users or targets the configuration commands and procedures may be grouped into a number of files as appropriate. Chapter 5 provides a worked example. This chapter concentrates on the commands used by `st20cc`.

The commands recognized by `st20cc` are listed in functional groups in the sections which follow and are identified by “`st20cc`” in the “*Tool environment*” field of each command definition in the "*ST20 Embedded Toolset Reference Manual*".

3.5.1 Start-up scripts

On start-up, `st20cc` will try to find and execute the scripts in the start-up command files described in section 1.4.1, before processing any command line arguments. This behavior may be turned off by the `-NS` command line option.

These start-up scripts may contain user-defined default command procedures. For example, if the following command:

```
include hello.cfg
```

was placed in the start-up file in the directory named by the environment variable `HOME`, then the command line in the example in section 3.5 could be modified as follows:

```
st20cc hello.c -p phone
```

3.5.2 Symbol handling commands

These commands perform standard symbol handling tasks.

Command	Description
`define`	Define a `const int` symbol
`entrypoint`	Define the main entry point of the program.
`export`	Specify the name of a function that is to be exported from the program.
`forward`	Create a forward reference to a symbol.
`import`	Specify the name of a function that is to be imported into the program.

The `define` command defines a constant integer variable with a given value. This can be used to avoid hard-coding constant values in the code of the program and to access segment and section information about the program. The `forward` command creates a forward reference to the specified symbol. This allows names to be made known to the linker in advance, or forces linking of library modules that would otherwise be ignored.

The `entrypoint` command specifies the main entrypoint for the program. If no main entry is defined an error will be returned. For C programs the supplied linker command start-up file defines the system main entrypoint, however, if the `st20cc -nolibsearch` option is used an entrypoint must be specified. A Relocatable Code Unit (RCU) must also have an entrypoint specified. The `st20cc -e` option also specifies a main entry point and is equivalent to `entrypoint`.

The `import` and `export` commands provide support for importing and exporting functions between applications. The commands are designed to be used with the functions provided in `dl.h`.

RCUs and the use of the import and export commands is described in the "*Building and running relocatable code*" chapter of the "*ST20 Embedded Toolset Reference Manual*"

3.5.3 File handling commands

These commands all perform file or directory handling.

Command	Description
`directory`	Add a directory to the search path.
`file`	Enable an input file to be specified to the linker.
`include`	Insert a command file.

The `directory` command adds a directory pathname to the file search path for command files, object files and library files. This command may be used in a command file but it must not be hidden within a command language construct such as a command procedure or an `if` or `while` loop.

The `file` command enables an input file to be specified, this may be an object file or a library file. Source files cannot be specified.

The `include` command enables another command file to referenced. The commands within the included file will be executed at the point the `include` command is specified. Any number of command files may be included within a command file and be nested to any level. `include` has the same restrictions as the `directory` command, that is, it must not be used within a command procedure or within other command language constructs.

3.5.4 st20cc options commands

Command	Description
`commandline`	Define a command line option for `st20cc`, see section 3.2.3.
`st20ccoptions`	Enable one or more `st20cc` command options to be specified in a command file.

This command enables `st20cc` command line options to be included in a command file. Any of the options in Table 3.1 may be specified.

3.5.5 Hardware configuration commands

This group of commands enables you to specify the target hardware or simulator.

Command	Description
`bootiptr`	Define the ROM boot address by setting up the initial value of the **Iptr** register.
`chip`	Define the chip type.
`emidelay`	Cause specified delay in EMI configuration for ROM systems.
`memory`	Define a memory segment.
`poke`	Write a value to an address.
`processor`	Define the processor core.

The `chip` command provides the processor core type based on the chip type. Additionally it defines memory segments of on-chip memory regions and declares variables in the created program that correspond to the addresses of on-chip peripherals.

The processor command enables the target ST20 processor to be specified and can be used instead of the `chip` command when the chip being used is not supported by the `chip` command.

The memory map is defined by specifying a number of `memory` commands so that each required memory segment is defined. (If the `chip` command has been used internal memory segments associated with the core will already be defined.) Once the memory map is defined it can be referred to by other commands such as those used to handle code and data placement. The memory command may also be used to reserve memory or perform absolute placement.

For example:

```
memory tracebuffer 0xc0000000 16 RAM
```

This reserves 16 bytes of RAM as a trace buffer, see section 7.3.10.

Note: that a memory segment of type `DEBUG` should be defined for use by the debugger.

The `poke` and `bootiptr` commands are only required when ROM output is to be produced, however, it is good practice to always include them. The `bootiptr` command defines the value of the **Iptr** at which the device starts executing. On ST20-C1 cores it must be specified and the **Iptr** must be in a ROM segment, word aligned and be the base of a memory segment. For an ST20-C2 core the **Iptr** must specify a half-word aligned address two bytes below the top of a ROM segment. The `poke` command is used to initialize peripherals such as an external memory interface (EMI).

The `emidelay` command allows insertion of delays in a ROM bootstrap sequence.

Creating a ROM image

When creating a ROM image, a ROM segment must be defined that includes the base address from which the device boots. Depending on the processor type this must be specified as follows:

- For an ST20-C1 (which has a variable ROM boot address) the base of the ROM segment must match the value specified for the `bootiptr` **Iptr**.

  For example:

  ```
  bootiptr 0x70000000.
  memory EPROM 0x70000000 (16 * K) ROM
  ```

- For an ST20-C2, the ROM segment must include the address `0x7ffffffe` that is, two bytes from the top of memory. For example:

  ```
  memory EPROM 0x7fff9C00 (25 * 1024) ROM
  ```

3.5.6 Rom file control commands

The commands in this group are all optional and modify a ROM image file.

Command	Description
`initstack`	Define the top address of the initial stack used by the ROM loader code.
`romimage`	Define properties of a ROM image file.

The `initstack` command specifies the top address of an initial small stack area which is used by the ROM loader code, before it moves to the main stack, specified by the `stack` command. If the `initstack` command is not specified then a default value is used. The `romimage` command enables you to override the default filename stem given to a ROM image file, to specify the base address of the ROM image and to specify that multiple output files are to be generated for a single segment.

For further details see the `initstack` and `romimage` command descriptions in the "*ST20 Embedded Toolset Reference Manual*").

3.5.7 Code and data placement commands

This group of commands are used to place sections of code and data in memory.

Command	Description
`addressof`	Return the address of the given memory segment or program section.
`bootdata`	Place the boot data block.
`heap`	Define the heap and optionally its size.
`place`	Place section in a named memory segment.
`stack`	Define the stack and optionally its size.
`sizeof`	Return the size of the given memory segment or program section.
`sizeused`	Return the amount used in the given memory segment.
`store`	Specify ROM segment that a section is stored in.

The `place` command is the principal command in this group and is used to place sections of code or data within a specified memory segment. The name of the memory segment is defined by a previous `memory` command. Sections are either defined using the `ST_section` pragma within the application source code or refer to default section names provided by `st20cc`.

The `heap` command is used to define the size of the heap, which is the area of memory that the C library function `malloc` uses. The `stack` command is used to define the size of the main stack (this is the stack, within which the application begins execution). The `stack` command is mandatory. If the `heap` command is not specified a zero sized heap is assumed.

The `store` command is optional and is only required if you wish to specify where sections, which reside in RAM at run-time, are stored in ROM before they are loaded into RAM. Where there are multiple ROM segments, `store` defines which ROM segment a specific section is stored in. By default all sections are stored in the first ROM segment encountered in the configuration procedure.

The `bootdata` command is also optional. The default is to place the boot data in either the largest segment of RAM or the first segment of ROM, as appropriate.

The `addressof`, `sizeof` and `sizeused` commands are all optional and enable information to be extracted about a specified memory segment.

The chapter “*Code and data placement*” in the "*ST20 Embedded Toolset Reference Manual*" describes how to define memory segments and place code and data within them. It also describes the default memory placements used if you do not specify any `place` commands.

3.5.8 System startup/shutdown support

The commands in this group are optional and define the order in which functions that have the `ST_onshutdown` or `ST_onstartup` pragmas specified will be invoked.

Command	Description
`endorder`	Define the order in which functions that have the `ST_onshutdown` attribute will be invoked.
`startorder`	Define the order in which functions that have the `ST_onstartup` attribute will be invoked.

Both functions take a priority argument, if a priority is not supplied the priority defaults to 0.

For functions where the `ST_onstartup` pragma is specified the most positive priority runs first and the most negative priority runs last. Functions for which the `ST_onstartup` pragma is specified but whose order is not specified using `startorder` will be called after all functions that have been ordered with the `startorder` command.

Behavior for `ST_onshutdown` is inverted, the most negative priority runs first and functions whose order is not specified will be called before all functions that have been ordered with the `endorder` command.

If `startorder` or `endorder` are not called at all the effective priority of a functions is in both cases more negative that the most negative integer supported.

3.5.9 Other commands

The following commands are also available when running `st20cc`:

Command	Description
`addhelp`	Add help for a procedure.
`cd`	Change current working directory.
`clinfo`	Turn on command language trace output.
`clerror`	Raise an error from a command language program.
`clsymbol`	Get information on command language symbols.
`eval`	Execute a command language procedure.
`fclose`	Closes open file.
`feof`	Tests for end-of-file marker.
`fgets`	Reads one line from a file.
`fopen`	Opens a file for reading, writing or appending.
`fputs`	Writes the specified string to a file.
`help`	Output help on command language.
`mknum`	Convert operand to a number.
`mkstr`	Convert operand to a string.
`mv`	Move a file.
`parse`	Execute string operand as a command language program.
`pwd`	Print current working directory.
`remove`	Remove a `when` or command language symbol.
`rewind`	Moves the file pointer to the start of the file.
`rm`	Remove a file.
`sys`	Execute a host command.
`write`	Send a message to the debugger output.

3.6 Order of linking

This section explains the order in which `st20cc` links object files and libraries. In particular it explains how the order that library files are presented on the `st20cc` command line can affect what is linked or even whether the linked unit is created successfully.

First some terms, for the purposes of this description:

- A module is a single object file generated by the compiler and may contain multiple sections. Usually this is synonymous with a `.tco` file.
- A symbol is a generic term for any object (data item or function).

3.6.1 Rules for linking

`st20cc` is the front end to `st20icc` (the compiler) and `st20link` (the linker) and simply calls these tools with the appropriate command lines. There are thus two levels of command line parsing for `st20cc` to perform.

`st20cc` applies the following general rules when linking:

- Object files (`.tco`) files and source files are included unconditionally.
- If any part of a module in a library is needed all of that module is linked.
- The libraries are searched in the order determined by st20cc, (see section 3.6.2 below).
- Standard libraries (for example, for `printf`, `strlen`.) appear last in the link (for example adding a `strlen` in a private library overrides the standard libraries supplied with the toolset).
- No library can contain duplicate symbols.
- Libraries (`.lib`) files specified on the command line are added to the list of libraries to be searched before configuration files are processed and therefore they are linked before any libraries referenced in a configuration file.

The search for any required symbols starts with the list of symbols already defined by the modules which have already been linked. If it is not found, then the list of symbols defined by all the libraries is searched, starting at the beginning of the list, and continues through all the modules and libraries until the symbol is found, or the end of the list is reached (in which case an error is reported). This may result in a previously unused module being linked in, in which case its symbols will be added to the list of defined symbols (checking for duplicates), and adds any unresolved symbols to the list of unresolved symbols. The search then start again with the next unresolved symbol.

3.6.2 Generating the link list

`st20cc` generates a list of modules to pass to the linker using the following rules:

- Object files (`.tco`) are placed first in the link list in the order they are specified on the command line.
- C and C++ source files that are compiled are placed in the link list, in the order they are specified on the command line.
- Library files (`.lib`) are placed after any object files in the link list, in the order they are specified on the command line.
- Configuration files (`.cfg`) appear on the link list in command line order after all other files.

Examples

```
st20cc -p c2 a.c b.c
```

In this example `st20cc` links `a.tco` to `b.tco` and then links the standard libraries.

```
st20cc -p c2 main.c a.lib b.lib
```

In this example `st20cc` links `main.tco` using the libraries `a.lib` and then `b.lib`.

```
st20cc -p c2 main.c a.lib -T b.cfg
```

This command line causes `st20cc` to link `main.tco` using `a.lib` followed by any libraries referenced in `b.cfg`.

Library examples

Consider the following example that uses the source files `main.c`, `one.c`, `two.c` and `three.c`, where:

`main.c`:

```
extern void a(void);
extern void b(void);

int main(void)
{
  a();
  b();
}
```

`one.c`:

```
#include <stdio.h>

void a(void)
{
  printf("function a in file one.c\n");
}

void b(void)
{
  printf("function b in file one.c\n");
}
```

`two.c`:

```
#include <stdio.h>

void a(void)
{
  printf("function a in file two.c\n");
}
```

`three.c`:

```
#include <stdio.h>

void b(void)
{
  printf("function b in file three.c\n");
}
```

To create the library files `one.lib`, `two.lib` and `three.lib` see “*Using the librarian tool*” in the "*ST20 Embedded Toolset Reference Manual*":

The following commands demonstrate how the order that these libraries are presented on the `st20cc` command line effect the link, map files are generated using the `-M` option to demonstrate which libraries are used to obtain symbols:

```
st20cc -p c2 main.c one.lib two.lib three.lib -M out1.map
```

This example links using the symbols a and b in `one.lib` and ignores the others (not needed).

```
st20cc -p c2 main.tco two.lib three.lib one.lib -M out2.map
```

This example links using the symbol a from `two.lib` and b from `three.lib` ignoring `one.lib` (not needed).

```
st20cc -p c2 main.tco two.lib one.lib three.lib -M out3.map
```

The link fails because the symbol a is supplied by `two.lib` and the symbol b is supplied by `one.lib` which also contains the symbol a, this clashes with the symbol a in `two.lib`.

The order of linking also has an impact on code size. Bringing in a module which contains many sections that are not needed, increases the code because the memory in a segment is filled from the bottom up and this causes an increase in the number of prefix commands required to achieve calls or jumps, this therefore affects the speed and sizeof the code.

3.6.3 Backward compatibility

The `-V18-cmdline-order` command line option can be used to force the command line arguments to be passed as they would have been under R1.8 and earlier toolsets. See section 3.2.2 for details.

3.7 Program build management

`st20cc` provides some simple program build management facilities. The `-depend` option will create a file containing makefile dependencies of the supplied source files. This file can be included into a makefile to ensure that build dependencies are always kept up to date.

The `-make`, and associated `-makeinfo`, provide a simple method of instructing `st20cc` to avoid unnecessary builds. If the `-make` option is given to `st20cc`, the tool will avoid recompilation of a file using the following rules:

- Does the object file exist?
- Is the source file dated later than the object file?
- Are any `#include` files dated later than the object file? (the `#include` file list is stored in the object file).

A similar process is used to avoid the creation of linked units, ROM images and libraries.

`-makeinfo` must be used in conjunction with `-make` in order to produce information about the make process. If the `-make` option is not used, then `-makeinfo` has no effect.

4 Support for C++

The ST20 Embedded Toolset currently provides only *minimal* support for C++, for example, templates are not supported. This chapter describes what support is provided by the toolset and its limitations.

4.1 Introduction

Using the ST20 Embedded Toolset, C++ programs follow a similar development path to C programs. The `st20cc` compile/link driver provides a single tool user interface to all stages of the C++ compilation and link, which the user can control via command line options.The C++ program can then be run and debugged via `st20run`, in a similar manner to a C program.

When processing C++ code, `st20cc` invokes the Edison Design Group preprocessor to convert C++ into C. This process is invisible.

The toolset's current implementation of the EDG 2.40 preprocessor supports the ISO C++ standard with the following omissions:

- Universal character set escapes.
- Try/catch around entire function.
- Templates.
- Standard Template Libraries.

In addition the throwing of exceptions is not thread-safe.

Note: because this toolset uses a preprocessor supplied by a third party which may change in subsequent releases, STMicroelectronics cannot guarantee C++ object and library compatibility. Users may have to re-compile C++ source code when migrating to subsequent releases of the toolset.

For further information on the C++ language, the reader is referred to the definitive text '*The C++ Programming Language*' by Bjarne Stroustrup, Addison-Wesley ISBN 0-201-88954-4.

The chapter "*C++ Language Features for the EDG preprocessor*" in the "*ST20 Embedded Toolset Reference Manual*" provides details of language implementation for the EDG preprocessor.

4.2 C++ driver

The compile/link tool `st20cc`, presents an integrated compiler interface to the user. For C++ programs it invokes the phases of a C++ program compilation in the correct order. Figure 4.1 illustrates how the compilation driver `st20cc` processes a typical C++ compilation command:

```
st20cc fileA.cxx fileB.icc fileC.c fileD.tco fileE.lib -p c2
```

where:

- `fileA.cxx` is a C++ module
- `fileB.icc` is a C++ preprocessed module
- `fileC.c` is a C module
- `fileD.tco` is an object module
- `fileE.lib` is a library
- `-p c2` calls the command language procedure `c2`, see Chapter 3.

The command produces the file `fileA.lku.`

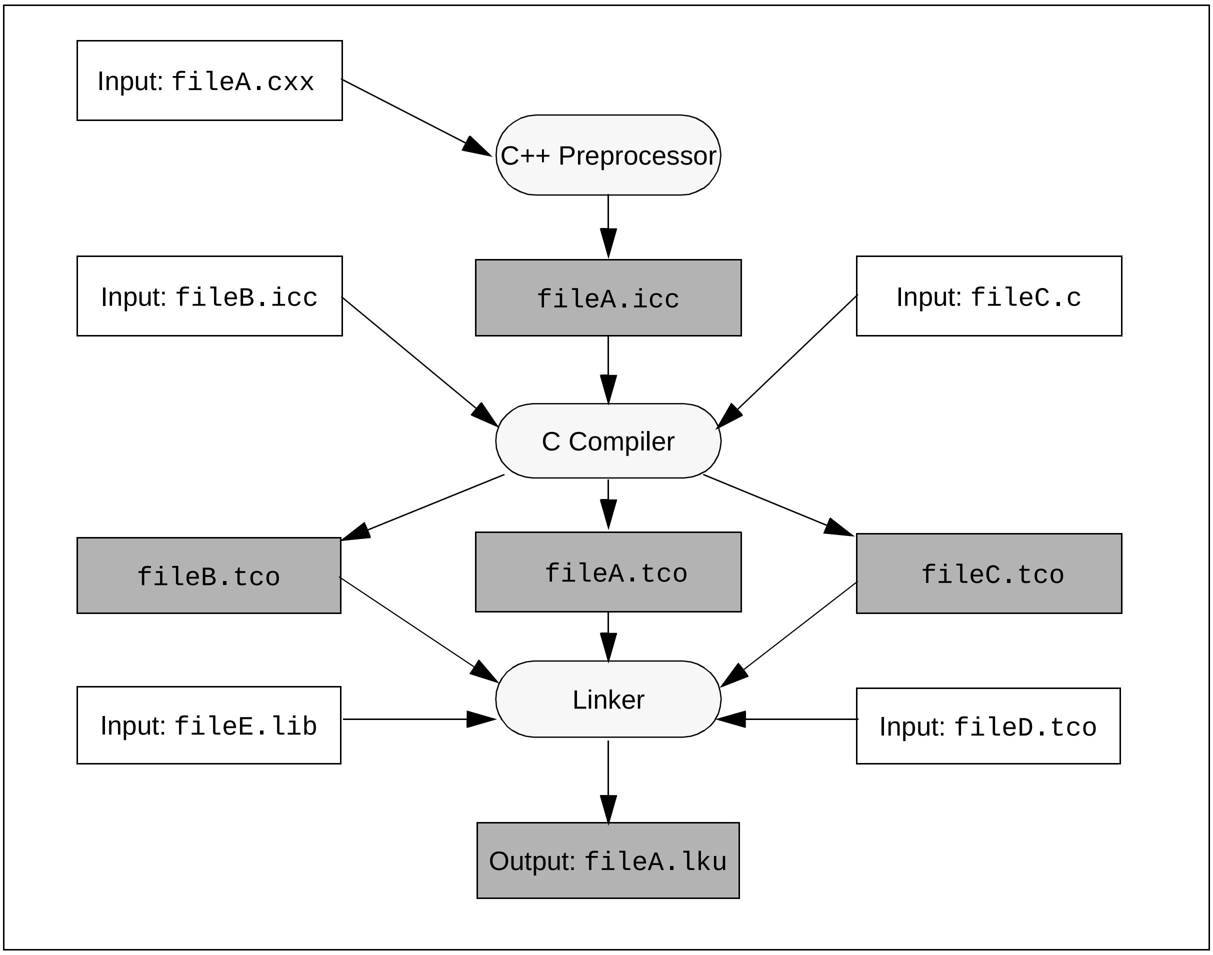


Figure 4.1 Phases invoked by the `st20cc` command

4.2.1 C++ preprocessor phase

The C++ preprocessor, `st20edg`, is the first phase invoked by `st20cc`. `st20cc` causes the C++ preprocessor to translate the C++ code into C code with filename extension `.icc`.

For example, compiling the file:

```
fileA.cxx
```

will generate the file:

```
fileA.icc
```

The C++ preprocessor places certain constraints on the preprocessing `#pragma` directive.The ‘*Preprocessing*’ chapter of the "*ST20 Embedded Toolset Reference Manual*" describes these constraints and a number of C++ specific preprocessing macros.

4.2.2 C compilation phase

The C compiler, `st20icc`, is invoked by `st20cc` to compile the intermediate C code, (generated by the C++ preprocessor), into object code. `st20cc` takes the C translated files (filename extension `.icc`) and passes them to the C compiler which produces an object module with filename extension `.tco`.

For example compiling the files:

```
fileA.cxx fileB.icc fileC.c
```

will generate the intermediate files:

```
fileA.tco fileB.tco fileC.tco
```

4.2.3 Linking phase

Once the compilation phase is complete, `st20cc` combines the C++ object modules and libraries to form a linked unit, resolving external references between modules. `st20cc` takes the object files (filename extension `.tco`) and the library files (filename extension `.lib`), and generates a linked unit. For example linking the files:

```
fileA.cxx fileB.ixx fileC.c fileD.tco fileE.lib
```

will generate the file: `fileA.lku`

4.3 `st20cc` command line

This section describes the support provided by the compile/link tool `st20cc` for C++. The `st20cc` command line syntax is as described in Chapter 3. Table 4.1 lists the additional input file types that are accepted to support C++.

File type	Description
`file.cxx`	A C++ source file.
`file.icc`	A C++ preprocessed file.

Table 4.1 C++ input files

Table 4.2 lists additional `st20cc` command line options which are provided to support C++.

Option	Description
`-F`	Run only the C++ preprocessor.
`-CXX=`*string*	Change the default extension of the C++ source files from `.cxx` to the string specified as argument of the option. For example: `st20cc -CXX=.xyz fileA.xyz`
`-diag-suppress` *list*	Suppress the warnings specified by number in *list*.
`-display-error-number`	Display error/warning numbers.
`-exceptions`	Exception handling enabled. Exception handling is not thread safe in this release of the toolset. By default exception handling is not enabled.
`-i`	Leave the intermediate `.icc` files in the current directory during the compilation process.
`-no-rtti`	Suppress run-time type information.
`-rtti`	Enable run-time type information.

Table 4.2 `st20cc` options for C++

Note that many of the standard `st20cc` command line options described in Chapter 3 are also applicable to C++ programs. Options not supported for C++, are marked with a '†' in Table 3.1.

4.3.1 Environment variables

One additional environment variable is supplied (`ST20EDGARG`) which may contain the default switches to the C++ preprocessor. For UNIX and Linux use, this should only contain spaces in the list of values if all the values are enclosed inside double quotation marks for example, `"value list"`. Preferably all quotes and spaces should be removed, for example:

```
setenv ST20EDGARG "-tused,-i"          (UNIX and Linux)
set ST20EDGARG=-tused,-i               (Windows)
```

4.3.2 TMPDIR directory full

`st20cc` creates a number of temporary files during the compilation process. It places these files in the standard UNIX and Linux or Windows temporary directory or in the directory specified by the `TMPDIR` (UNIX and Linux) or `TMP` (Windows) environment variable. If there is not enough space in the directory, the compilation will be aborted.

4.4 Libraries

The C++ libraries provide library support for the C++ preprocessor. They consist of the following:

- `libC.lib`. C++ Standard Library (excluding I/O Stream Library).
- `libCx.lib`. C++ Standard Library (excluding I/O Stream Library) for use when exceptions are enabled.
- `libCnew.lib`. C++ New Library.
- `libCnewx.lib`. C++ New Library for use when exceptions are enabled.
- `libCvirt.lib`. C++ Virtual Function Library.

The Standard Template Libraries are not included in this toolset release.

4.5 Debugging C++

This section describes the support provided by the toolset for debugging C++ programs. This includes extensions to the toolset command language as well as support provided by `st20run` and the debugging GUI.

4.5.1 Class Member Variables

Access to the member variables of a C++ class can be achieved, using the command language `print` command, in the same way that C structures can be accessed.

Variable access can also be augmented with class information in the same way as in C++.

For example:

If the source code contains the classes:

```
class fruit {
public:
  char *name;
  int   colour;
};

class apple : public fruit {
public:
  char *name;
};

apple a;
apple *aptr = &a;
```

the values of the variables used can be printed by using the `print` command:

```
> print a
> print *aptr
> print a.name
> print a->name
> print a.colour
> print a.fruit::name
> print a->fruit::name[3]
```

When `print` lists the contents of a class it will show the member variable and also the class to which it belongs:

```
> print a
{
  fruit::name
  fruit::colour
  apple::name
}
```

4.5.2 Class Member Functions

Function references used by the command language refer to member functions by using C++ style notation:

```
> break apple::identify          ## break on apple::identify()
> addressof apple::identify      ## print the address of apple::identify()
```

4.5.3 Global Variable Scoping

Preceding a variable name with a double colon ‘`::`’ indicates that the global variable is required.

```
> print ::globalvar
```

4.5.4 Member Variable Scoping

When finding the value of a variable the debugger follows the same scoping rules as C++. Therefore when stopped inside a member function, giving the name of a member variable will provide the value of that member variable, except when that variable name has been superseded by a local variable. Assuming we have stopped in `apple::identify(void)` then

```
> print name              ##print the member variable name
“apple”
> print fruit::name       ##print the name member of the fruit class
“fruit”
```

4.5.5 Overloaded Functions

C++ allows a function to be defined more than once with different numbers and types of parameters. The debugger handles these functions by including the parameter list with the function name. It is therefore possible to set a breakpoint on one version of a function by specifying the parameter list to select the correct version:

```
> break “func(int,char*)”
```

When specifying functions in this way it is necessary to enclose the expression in double quotes, or the interpreter will misinterpret the expression. Due to the way the function symbols are stored it is also necessary to exactly match the stored symbol when selecting functions in this way. It can be helpful to use the symbols command to show a list of suitable matches to show exactly how the symbols are stored, however, it is much easier to use the `break` command to list all possible versions of the function.

If the function specified to the break command is unique then a breakpoint is set. If the function specified is overloaded and only the function name is specified then an error occurs, no processing takes place and a list of possible matches is produced:

```
> break func
func\1 => func(int)()
func\2 => func(int,char*)()
func\3 => func(int,int,char*)()
```

It is then possible to set a breakpoint on one of these functions by specifying the function name and the version number:

```
> break func\2
```

4.5.6 Nested Classes

The variable and class scoping can also be extended for use in nested classes:

```
class enclose {
public:
  class inner {
   public:
     int h;
     void print(void);
   };
};
```

If we have stopped in `enclose::inner::print(void)`

```
> print h                    ## print member h
20
> print enclose::inner::h    ## print member h of the inner class
20                           ## which is in turn a member of the enclose class
```

4.5.7 References

The compilation chain of this toolset causes the C++ source to be precompiled into C. This has the side effect of implementing all the reference variables as pointers, and it is therefore impossible for the debugger to know what is a pointer and what is a reference. Hence, all reference variables will act like pointer variables as far as the debugger is concerned.

4.5.8 Class Static Variables

When a class is defined as containing a static member variable this variable is not actually included in the structure definition of the class. It must be defined separately:

```
int classname::staticvar = 0;
```

Access to this variable is then only possible via that name:

```
> print classname::staticvar
```

4.5.9 Not supported

Support is currently not available for:

- Class breakpoints
- Breakpoints on specific instances of classes
- Pointer to member operators (.* and ->*)

5 Defining target hardware

A target system needs to be described, including its memory size, core type and which diagnostic controller unit (DCU) it uses for debugging, before code can be built for it by `st20cc`, or loaded by `st20run`. This information is given to the tools as a command language procedure containing a series of command language commands. (The command language is described in the "*ST20 Embedded Toolset Reference Manual*".)

Parts of the target descriptions are used by the compile/link tool `st20cc`, the application load/debug tool `st20run` and the simulator `st20sim`. To build the code, `st20cc` needs to know the type of target, in order to link in the correct libraries. It also needs to know where the memory is, in order to place the code and data space correctly. To run the application, `st20run` needs to know how to initialize the target hardware, how to load the code onto the target, how to interact with the running code, which DCU is used by the silicon, and where to find the code and data. When an application is run on a simulator, the simulator needs to know which core it is simulating and where the memory is.

The commands listed in Table 5.1 can be used in a command language procedure to define target characteristics.

Command	Description
`bootiptr` *iptr*	Define the initial **Iptr** of an ST20C1 core.
`chip` *type*	Define the type of the chip variant.
`memory` *name base size type*	Declare a memory segment for the current target connection.
`poke` *address value*	Write to the memory of the current target connection.
`processor type`	Define the type of processor on the target
`register name` *address options*	Associate symbolic name to peripheral register address.
`reset`	Reset the current target connection.

Table 5.1 Target configuration commands

The commands used by `st20cc` to define the application software or target are called configuration commands. Descriptions of targets are saved as command language procedure definitions, each containing the configuration commands to describe the target. These procedures are normally referenced on the command line of `st20cc` and `st20sim` using the p option, or by `target` commands in `st20run`.

These procedure definitions are normally saved in a configuration file which will act as a database of available targets. By convention, configuration files are given a `.cfg` extension. Configuration files are used for linking, in order to place the code and data in hardware memory segments. The same files may also be used for running and debugging the application, telling `st20run` how to initialize and interact with the target. When running on a simulator, the configuration is used to notify the simulator of the hardware it is simulating. The tools will ignore any commands which they do not support.

5.1 Defining target characteristics

A target configuration typically includes:

- The type of chip variant, defined by the `chip` command, for example:

  ```
  chip ST20TP3
  ```

  Used by `st20run` to provide variant specific capabilities such as processor type, DCU, peripheral memory definitions, register definitions and a variant-specific trap handler (to keep watchdogs alive, for example). This is used by `st20cc` to define standard characteristics of processors such as the processor type, internal memory and peripheral memory. See section 5.1.1.

  The toolset needs to know which diagnostic controller is being targeted because the base addresses, trap handler and inform routines used are different. The toolset identifies the type of DCU from the `chip` command. If no `chip` command is executed then by default the toolset builds for and expects to find a DCU2 diagnostic controller. Building for a DCU3 is described in section 5.4.

- Alternatively, the type of processor, defined by a `processor` command, for example:

  ```
  processor c1
  ```

  Used by `st20cc`, `st20run` and `st20sim` to determine which core to compile and link for, or execute on, or simulate, when the chip is not supported by the `chip` command.

- The amount, type and location of memory and memory mapped devices on the target board, each segment defined by one `memory` command, for example:

  ```
  memory EXTERNAL 0x4000000 (4 * M) RAM
  ```

  Used by `st20cc` when placing sections and by `st20run` and `st20sim` to understand the memory layout of the target. Each target configuration must provide one memory segment for `st20run` to load the breakpoint trap handler into. This memory segment has the type `DEBUG` and name ‘`TRAPHANDLER`’ and must be at least 1024 bytes in size, for example:

  ```
  memory TRAPHANDLER 0x40000000 (1 * K) DEBUG
  ```

  Note: that the `chip` command will define an INTERNAL memory segment.

- Any initialization of memory or memory mapped devices, using `poke` commands. These commands are used by `st20cc` to initialize the target when a ROM image is generated. They are also used by `st20run` to initialize the target so it is suitable for application load and interactive debug, for example;

  `poke` *0x00002060 1 -device*

  If the `-device` option is set, a device write is performed. Device writes are required when accessing on-chip devices of ST20-C2 core based chips.

- Any register definitions for the target, defined by the `register` command, for example:

  ```
  register DEV1STATUS 0x50000000 -group DEV1
  ```

  Used by `st20run` when displaying registers.

- For ST20-C1 cores the starting value of the **Iptr** register should be defined using a `bootiptr` command.
- In order for `st20run` to load and debug an application, the target board must be reset using the `reset` command, unless the system being debugged is already connected and running.

5.1.1 ST chip and board definitions

The `stdcfg` release directory contains many useful default configuration procedures which are automatically pulled in by the toolset. For example, using the `chip` command will automatically access one or more of these files.

The board release directory also contains procedures that define various ST20 evaluation boards. To use these procedures, add the following line to your configuration file:

```
include boardprocs.cfg
```

This may be added to a start-up file as shown in section 1.4.1.

5.2 Data caches in internal SRAM

On some ST20 devices the memory used by the data cache can be treated as extra internal SRAM if the data cache is not enabled. See your device datasheet for details.

On such devices the use of the data cache effectively reduces the amount of internal SRAM available to the application. However, the processor is not prevented from writing to this memory which can cause cache corruption. It is therefore important that the data cache memory is protected from being used by the application by reserving it during the link process. The recommended method of doing this is to call the command procedure "`dcache_internal`". This will automatically reserve the top 2K of internal memory. For example:

```
proc sti5500board_dcache {
  chip STi5500
  dcache_internal
  ....
}
```

For devices that do not have this feature, do *not* call `dcache_internal` as this will waste internal memory.

5.3 Worked Example

For example, consider a target system with a STi5500 (ST20-C2 core) with 4 Kbytes of internal SRAM starting at address 0x80000000 (the first usable address of which is 0x80000140), 2 Mbytes of external DRAM starting at 0x40000000, some external devices starting at 0x50000000 and 4 Mbytes of ROM starting at 0x7fc00000 as shown in Figure 5.1:

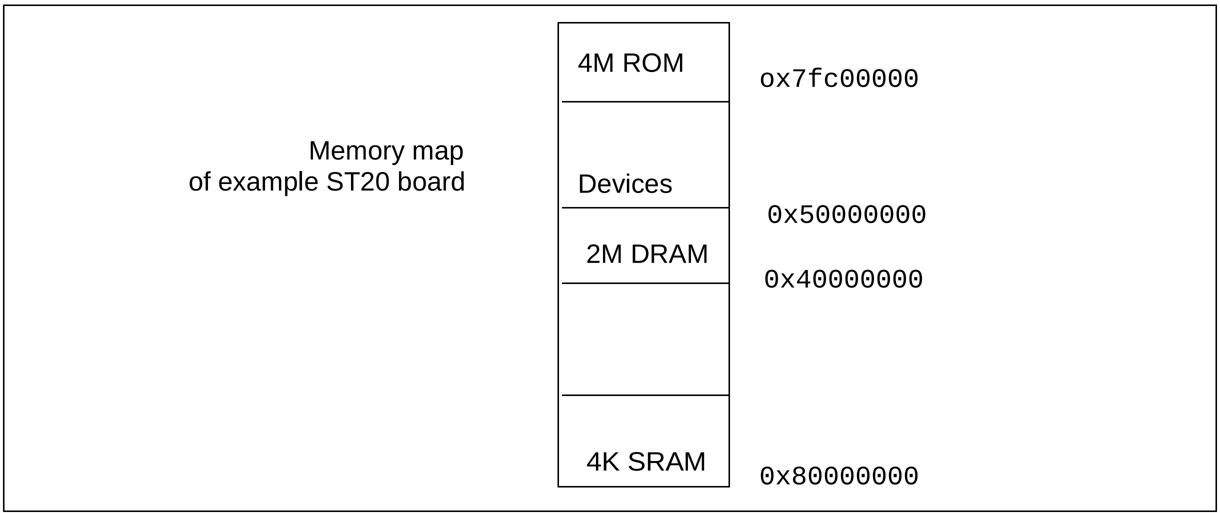


Figure 5.1 Target `st20board` memory map

The following command language procedure definition can be used to describe the hardware characteristics of the example system. The procedure is called `STi5500board`:

```
proc STi5500board {
  chip STi5500
  memory EXTERNAL 0x40000000 (2*M) RAM
  memory TRAPHANDLER (0x40000000+2*M-1*K) (1*K) DEBUG
  memory DEVICE1 (0x50000000) (1*K) DEVICE
  memory DEVICE2 (0x50000100) (16) DEVICE
  register DEV1STATUS 0x50000000
  memory FLASH (0x7fc00000) (4*M) ROM
}
```

Note: that in this example the `chip` command will define a memory segment for internal memory (called `INTERNAL`) and that the debug segment is declared to be within the `EXTERNAL` memory segment.

Note: a target system which has a memory mapped data cache that is in use, should reserve the top 2 Kbytes of the `INTERNAL` memory segment for the data cache. This will prevent code or data being placed in the memory region used by the data cache.

The following example shows how the procedure "`dcache_internal`" (found in the supplied configuration procedure `cache.cfg`) can be used to define a segment used exclusively by the data cache:

```
proc STi5500board_dcache {
   STi5500board
   dcache_internal
}
```

To enable `st20run` to load and run programs any ST20 external memory interface (EMI) devices must be initialized by `poke` commands, because the external memory cannot be accessed until the EMI is initialized. The commands can be defined in a different procedure.

```
proc STi5500emi {
  poke -d 0x00002000 0x00000570
  poke -d 0x00002004 0x0000fff0
  poke -d 0x00002008 0x0000ff46
  ...
  poke -d 0x00002038 0x00004006
  poke -d 0x0000203c 0x00000000
  poke -d 0x00002060 0x00000001
}
```

Other target specific initializations, such as peripheral setup, can be performed using poke commands and will normally be written as peripheral specific initialization procedures.

To use `st20run` to load and debug the target, the target board must be reset using the `reset` command and the system memory of the processor should be initialized by calling the procedure `st20c2MemoryInit`. This reset, together with calls to the previous procedures can be put into a further procedure that when executed has a complete description of an initialized target.

```
proc hw5500 {
  reset
  st20c2MemoryInit
  sti5500board
  sti5500emi
}
```

The procedure can be invoked by `st20run` by associating it with a target using the `target` command, for example:

```
target eval5500 tap "jei_soc chitty" hw5500
```

These procedures and the target definition can be put into a single configuration file `sti5500.cfg`. This file can be supplied explicitly to `st20run` using the `-i` option or alternatively it can be included into one of the command language startup files, for example:

```
>st20run -i sti5500.cfg -t eval5500
```

When `st20run` connects to `eval5500` (using the `st20run -t` option or the `connect` command) it will invoke the procedure. The same procedure can be used to define the characteristics of a simulation. This is done by creating a simulated target that invokes the procedure.

For example:

```
target eval5500sim st20sim "st20sim -q -f sti5500.cfg -p
hw5500" hw5500
```

Target definitions procedures and st20cc

The procedures that are used by `st20run` (and `st20sim`) that define the target can be used by `st20cc` in the process of building programs. To build a linked unit (`.lku` file) that can be loaded by `st20run` an additional command procedure is required that defines the stack and heap requirements of the program.

For example:

```
proc link5500 {
  hw5500
  heap EXTERNAL (100*K)
  stack EXTERNAL (20*K)
}
```

This can be used by `st20cc` using the `-p` option:

```
>st20cc -T sti5500.cfg -p link5500 hello.c -g
```

This will produce the program `hello.lku` that can be run on the target hardware using:

```
>st20run -i sti5500.cfg -t eval5500 hello.lku
```

or onto a simulation of the target using

```
>st20run -i sti5500.cfg -t eval5500sim hello.lku
```

A ROM image can be built by extending the linkage procedure to define placement of the code into the ROM segment:

```
proc link5500 {
  ...
  place def_code FLASH
}
```

`st20cc` is instructed to build a ROM image using the following command:

```
>st20cc -T sti5500.cfg -p link5500 hello.c -off hex -g
```

The ROM image that is created will contain the information defined by the poke commands (in `sti55emi`) to enable the EMI to be programmed.

5.4 Building code for the DCU3

Users must use the `chip` command before linking and running code on devices with DCU3 diagnostic controllers. This ensures that the correct DCU type and base address are used.

For example, to build a program `myprog.c` for a STi5514, the commands might look like the following:

```
proc myboard {
  chip STi5514
  memory EXTERNAL 0x40000000 0x10000 RAM
  stack EXTERNAL
  heap EXTERNAL
}
st20cc -T mydemo.cfg -p myboard myprog.c
```

The set of command file procedures utilized to build and run on a DCU3 part is different from the command file procedures for a DCU2 part. An existing application built for a DCU2 part must therefore be linked again to run on a DCU3 part.

6 Interfacing to the target

This chapter describes the different types of possible targets, including simulators that can be connected to the host. The chapter describes how to use the toolset command language to create a target definition.

This toolset supports the following target types:

- ST20 with ST20-C1 or ST20-C2 core and a diagnostic controller unit (DCU) on chip.
- ST20 simulator.

In this chapter, the term *target* is used to mean any software simulator or hardware which can run an ST20 application.

This toolset supports connections from the following hosts:

- a Sun workstation running Solaris 2.5.1 (or later) and X-Windows;
- a PC running Windows 95/98/2000/NT;
- a PC running Red Hat Linux V6.2.

A hardware target will generally be a ST20 development board with a JTAG test access port (TAP) connected to an on-chip Diagnostic Control Unit (DCU). There are several methods of connecting the development board to the host:

- via an interface connected to a host parallel port;
- via an interface connected to an Ethernet network;
- via an interface connected to a host Universal Serial Bus port (USB).

This chapter describes how to set up the target descriptions. Descriptions of the named targets may be held in files, so that once the files are set up, the named targets can be used by many applications without target-specific coding. The descriptions are in the form of command language commands.

Installation information relevant to the interfaces is given in the '*ST20 Embedded Toolset Delivery Manual*'.

6.1 The target command

The `target` command is used by `st20run` to declare a target name and defines how the target is accessed, that is the interface to the target. For hardware targets, this uniquely defines a physical target. The details of the target interface description depend on the type of interface and are described below. The target command used without any arguments will list the defined targets.

`st20cc` and `st20sim` ignore the target command.

The `target` command syntax to define a target is:

`target` *name interface_type interface_arguments* [*command*] where: the target *name* is the name given in the `connect` command to define the target currently being used.

The *interface_type* and *interface_arguments* define how `st20run` will access the target. Some possible values are given in Table 6.1. These are described in more detail below.

The command language statement *command* will be executed to initialize the target when it is connected using the `connect` command.

For example, the following target commands define one simulator target called `oursim` and one hardware target called `st20`.

```
target oursim st20sim "st20sim -q -f st20hw.cfg" mysim
target st20   tap     "jei_soc  miracle"      st20boardhw
```

Value of *interface_type* **and** *_arguments*	**Type of interface**
`st20sim "`*simulator_command_line*`"`	ST20 simulator. See section 6.7.
`tap "jei_soc `*IP_address*`"` or `tap "jei_soc `*host_name*`"`	ST20-JEI or ST Micro Connect connecting Ethernet to an ST20 core test access port. (Windows, UNIX and Linux). See section 6.2.
`tap "jpi_ppi `*parallel_port_device_name*`"`	ST20 -JPI connecting parallel port *device_name* to a ST20 core test access port. (**Windows Only**). See section 6.6.
`tap "hti_ppi `*parallel_port_device_name*`"`	ST Micro Connect connecting parallel port *device_name* to a ST20 core test access port. (**Windows Only**). See section 6.4.
`tap "hti_usb `*usb_port_device_name*`"`	ST Micro Connect connecting USB port *device_name* to a ST20 core test access port. (**Windows Only**). See section 6.3.

Table 6.1 Interface types

The `target` commands may be listed in a targets file. The targets file is normally referenced by an `include` statement in the `st20run` start-up file, as described in section 7.1.2. The configuration procedures should be defined before the target, so the targets file is normally of the form:

```
# Targets file
include hwconfig.cfg
target sim  st20sim "st20sim -q -f st20hw.cfg"  mysim
target st20 tap     "jei_soc miracle"       st20boardhw
target st20micro tap "jei_soc 138.198.7.25" myboard
```

Chapter 7 describes how to run or debug an application on a defined target.

Each physical target can be given any number of names, and any number of physical targets can share the same name. An application may be run using a shared target name, in which case the first free physical target with that name will be selected.

For example:

```
target st20 tap "jei_soc  miracle1" st20boardhw
target st20 tap "jei_soc  miracle2" st20boardhw
target st20 tap "jei_soc  miracle3" st20boardhw
```

You can now use the target name `st20` for an application which can be run on any of the target definitions.

6.2 ST Micro Connect and ST20-JEI Ethernet connection

ST Micro Connect and ST20-JEI allow connection from an Ethernet based UNIX, Linux or Windows host to a single JTAG based ST20 development board. Once connected, the development target can be accessed by the toolset.

ST Micro Connect or ST20-JEI should be connected to the Ethernet network, as shown in Figure 6.1.

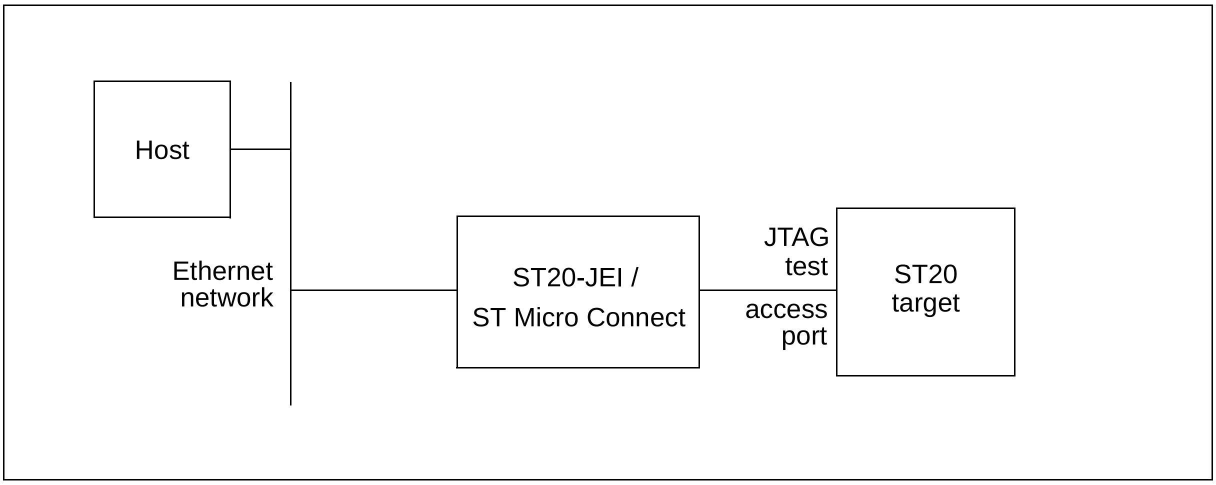


Figure 6.1 Interfacing to Ethernet via ST20-JEI

A hardware target must be of type `tap`. This signifies an ST20 Test Access Port, in this case connected to the host through an ST Micro Connect or ST20-JEI interface, as shown in Figure 6.1.

For Ethernet networked hardware targets, connected via an ST Micro Connect or ST20-JEI, the interface arguments are:

`"jei_soc` *IP_address*`"` or
`"jei_soc` *host_name*`"`

where `jei_soc` signifies that an ST Micro Connect or ST20-JEI interface is being used and *IP_address* or *host_name* maps to the internet protocol (IP) address of the ST Micro Connect or ST20-JEI.

For example, if the target is an ST20 connected via an ST Micro Connect or ST20-JEI which has the IP address `138.198.7.25`, then the target command is of the form:

`target` *name* `tap "jei_soc 138.198.7.25"` *config_proc*

where *name* is the name being defined for the target and *config_proc* is the command language target configuration procedure defining and initializing the target.

6.3 ST Micro Connect USB connection

The ST20 toolset supports connections to ST Micro Connect over USB (Universal Serial Bus) under Windows 98 and Windows 2000. ST Micro Connect should be connected to the PC, as shown in Figure 6.2. For more information about installing ST Micro Connect, please refer to the '*ST Micro Connect datasheet*' - *ADCS 7154764*. **Note:** this method of connecting to a PC is not available to Linux hosts.

A hardware target must be of type `tap`. This signifies an ST20 Test Access Port, in this case connected to the host through the ST Micro Connect USB interface, as shown in Figure 6.2.

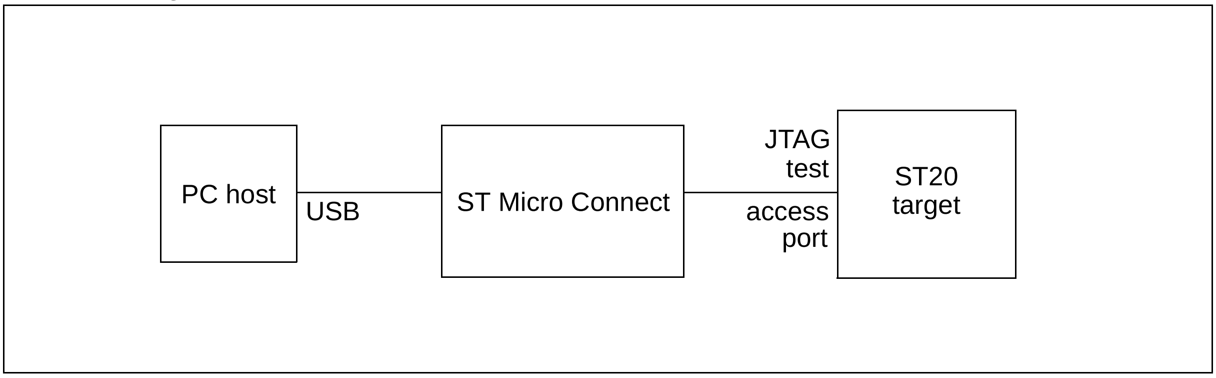


Figure 6.2 Connecting to ST Micro Connect over USB

The interface argument for this type of interface is of the form:

`"hti_usb` *ST_Micro_Connect_USB_device_name*`"`

where `hti_usb` signifies that the ST Micro Connect is directly connected to the PC's USB interface. *ST_Micro_Connect_USB_device_name* refers to one of the following:

- The reserved name `"usb"` which signifies the ST Micro Connect interface that is currently connected to the USB interface. This is the simplest option, but can only be used if there is only one ST Micro Connect connected to the USB interface. If there is more than one device connected it will need to be identified using either the unique name or the serial number described below.
- The unique name allocated to the ST Micro Connect interface when first connected to the PC, for example: `HTI1`, `HTI2`. This name can be determined by connecting the ST Micro Connect to the USB interface and starting up the Windows Device Manager. The connected ST Micro Connect interfaces are listed in the "*ST Debug Interfaces*" section. The ST Micro Connect's name and serial number are listed in the device's properties. The name given to a particular ST Micro Connect interface will always be the same on a particular PC, but maybe different on other PCs.
- The serial number of the ST Micro Connect interface. This is actually the device's Ethernet address which can be found on the case or in the Windows Device Manager properties dialog described above. The serial number will always be the same for a particular ST Micro Connect interface, irrespective of the PC it is connected to.

For example, if the target is an ST20 connected via an ST Micro Connect which is itself the only ST Micro Connect connected to the PC's USB interface, then the `target` command is of the form:

`target` *name* `tap "hti_usb usb"` *config_proc*

where *name* is the name being defined for the target and *config_proc* is the command language target configuration procedure that defines and initializes the target.

If the device has the unique name `HTI1` and the serial number `00:80:e1:42:02:01`, then the target command could be either of the following:

`target` *name* `tap "hti_usb hti1"` *config_proc*
`target` *name* `tap "hti_usb 00:80:e1:42:02:01"` *config_proc*

6.4 ST Micro Connect Parallel port connection

The ST20 toolset supports connection to a single ST Micro Connect over a parallel port. Supported modes are ECP mode, nibble mode and Byte mode. ST Micro Connect should be connected to the PC, as shown in Figure 6.3. The parallel port driver is installed as part of the ST Micro Connect installation, refer to the 'S*T Micro Connect datasheet*' - *ADCS 7154764* for details. **Note:** this method of connecting to a PC is not available on Linux hosts.

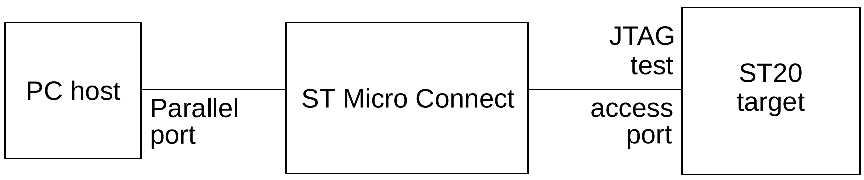


Figure 6.3 Connecting to ST Micro Connect over Parallel Port

A hardware target must be of type `tap`. This signifies an ST20 Test Access Port, in this case connected to the host through the ST Micro Connect parallel port interface, as shown in Figure 6.3

The interface argument for this type of interface is of the form:

`"hti_ppi` *parallel_port_device_name*`"`

where `hti_ppi` signifies that the ST Micro Connect is directly connected to the PC's parallel port interface.

For example, if the target is an ST20 connected via an ST Micro Connect to the parallel port `LPT1`:, then the `target` command is of the form:

`target` *name* `tap "hti_ppi lpt1"` *config_proc*

where *name* is the name being defined for the target and *config_proc* is the command language target configuration procedure that defines and initializes the target.

6.5 ST20-JEI and STMicro Connect trouble-shooting

For the ST20-JEI and ST Micro Connect the `target` command provides the ability to set the clock speed of the JTAG interface. It is possible for the JTAG connection to run faster than the target hardware can cope with, this is signified by the messages:

```
target not responding
communication timed out
```

The clock speed of the JTAG interface is set by adding an assignment to the `tckdiv` parameter in the target definition. The JTAG clock frequency is divided by this parameter. This parameter takes powers of 2 ranging from 2^0 to 2^{22}. A value other than a power of 2 is rounded up to the next power of 2. The default value is 1.

As an example the following target definition instructs the ST20-JEI `myjei` to set its clock frequency to run at half the default speed:

```
target myjei tap "jei_soc myjei tckdiv=8" "reset;tp3"
```

6.6 ST20-JPI Parallel port connection

This section provides guidance on configuring, trouble-shooting and connecting to the PC/ST20-JPI parallel port interface. Connecting to a parallel port is supported only for PC Windows hosts.

An ST20-JPI parallel port interface connects the TAP to the parallel port of the host PC, as shown in Figure 6.4.

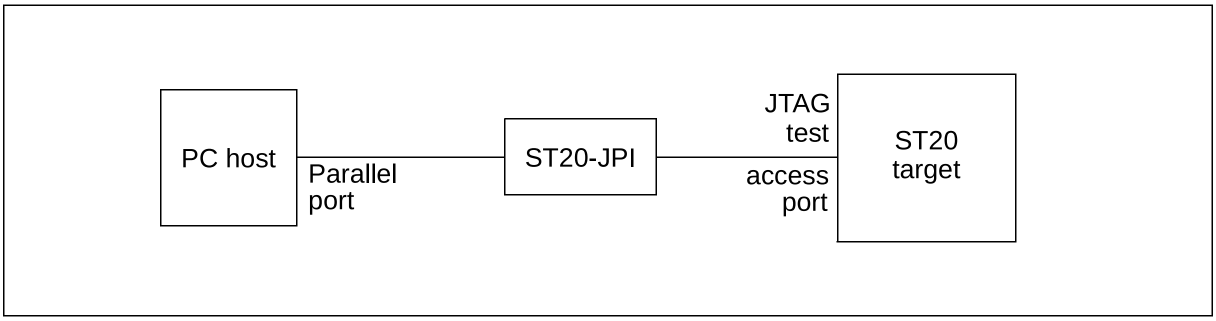


Figure 6.4 Connecting to a PC parallel port

6.6.1 PC Parallel port modes

There are a number of modes a PC's parallel port can be put into, to achieve the bi-directional communications required by `st20run`. The supplied parallel port driver only supports a subset of them; EPP, byte and nibble modes. The mode is usually selected from the PC's BIOS setup screen; to use the driver you must select the appropriate mode for your PC.

The most common modes you will see are as follows:

- 4-bit or nibble mode is the most widely supported parallel port mode. It should be possible to use it on most PCs, however, it provides the lowest performance of the all the modes. It is sometimes referred to as "AT" or "compatible" mode on some PCs.
- 8-bit, byte, or bi-directional mode is available on most modern PCs. It provides better performance than 4-bit mode.
- EPP mode is available on most modern PCs. It provides the highest performance of the modes supported by the parallel port drivers.
- ECP mode is not supported by the ST20-JPI. Modern PCs tend to have this enabled by default. EPP, byte or nibble mode must be selected before the parallel port can be used.
- PS/2 mode is not supported by the ST20 toolset's parallel port drivers.

6.6.2 Driver configuration using `vppiset.exe`

Certain driver parameters can be modified either for diagnostic or tuning purposes. A tool called `vppiset.exe` has been provided with the toolset to simplify the setting of these parameters. Figure 6.5 shows `vppiset.exe` running with the default parallel port options selected.

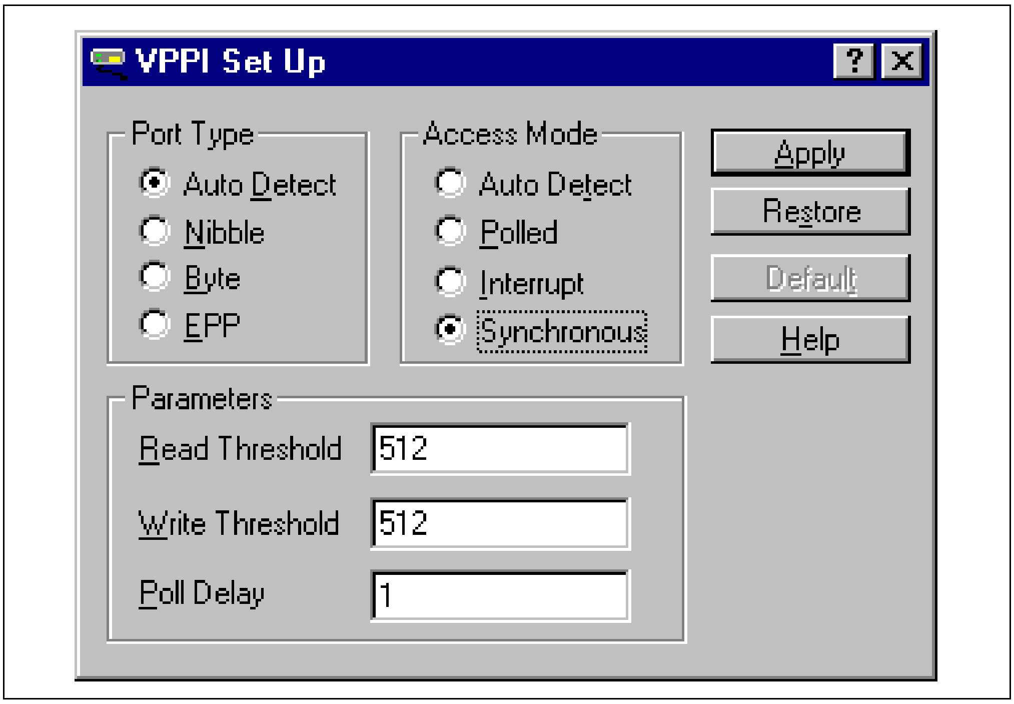


Figure 6.5 `vppiset.exe` main window

The program allows the following settings to be defined:

- Port type selects the parallel port mode to use. The default setting is “Auto Detect”. If “Auto Detect” is selected the driver will attempt to choose the optimum mode for your PC (see section 6.6.1 for more information). If any of the other modes are selected, the driver will be forced to use that mode and will not detect if it is compatible with the PC hardware.
- Access mode tells the parallel port driver how to access the port. The default setting is “Synchronous” which provides the best performance without using interrupts. **Interrupt mode can improve performance on many PCs, however, STMicroelectronics has found that some machines do not conform to the IEEE 1284 parallel port specification which may result in unreliable behavior**. **Problems can occur sporadically, often on an apparently working system. We suggest that if you experience communications related problems such as the system stopping or time-outs you revert to synchronous mode.**
- The read and write thresholds set the maximum number of bytes transferred in a single burst. Increasing this number may improve parallel port performance; reducing it will improve operating system responsiveness.
- Poll delay sets the period of time in milliseconds to sleep between each poll of the parallel port. The default value for this is 1 millisecond which is the smallest interval possible. Increasing this value will improve operating system responsiveness at the cost of reduced parallel port performance.

The settings are not used by the driver until the “Apply” button is activated. Use the “Restore” button to revert to the current used settings. The “Default” button reverts to the default settings.

6.6.3 Trouble-shooting

The parallel port driver attempts to determine the optimum configuration for your system. However, if `st20run` will not connect to a ST20-JPI it is a good idea to try out the lowest performance parallel port modes first. The mode can either be set from the PC’s BIOS setup screen or the drivers can be forced to use a particular mode. See section 6.6.1 and section 6.6.2.

If `st20run` rejects a connection with a “`cannot connect to target...`” message this indicates that it cannot talk to the ST20-JPI at all. This could be caused by:

- An ST20-JPI with no power.
- A bad connection between the PC and the ST20-JPI.
- An ECP mode Windows driver. If Windows 95 has installed an ECP device driver rather than the normal printer port driver the `st20run` ST20-JPI driver will reject the connection. To overcome this, use the System/Device Manager to change the driver.
- Target not specified in configuration file.

A “`cannot boot...`” or “`target is not responding`” error from `st20run` could be caused by:

- Interrupt problems. STMicroelectronics has found that some PCs operate outside the IEEE 1284 parallel port specification which means that interrupts could be missed at random points in communication. If this is the case we suggest that the parallel port driver is forced into synchronous mode (see section 6.6.2).
- Incorrectly set up interrupts caused by an IRQ clash with another device. See section 6.6.2 for details on enabling synchronous access mode.
- Excessive external interference. Certain electrical devices such as televisions or power supplies in close proximity to the ST20-JPI have been found to cause reliability problems.
- An EPP device driver from the OS-Link toolset already installed (Windows 95/98 only). The OS-Link toolset’s EPP driver is not compatible with the one used in the DCU toolset. To remove this driver from the system, the line:

  ```
  device=c:\st20swc\tools\vb045st.386
  ```

  should be removed from the `[386Enh]` section of the `C:\Windows\System.ini` file.
- “Noisy” parallel port interface card. STMicroelectronics has found that some parallel ports (mainly parallel port interfaces integrated onto the motherboard) produce interference that the ST20-JPI cannot cope with. A revised ST20-JPI is available (DB221A.2) which has a fix for this problem.

If there does not appear to be a significant performance advantage between using the parallel port driver in polled mode and interrupt driven EPP mode then the problem could be:

- The driver defaulting to polled mode. If the driver fails to detect an interrupt it will default to polled mode. This can be caused by another driver being installed that has already claimed the IRQ or an incorrectly configured parallel port. See section 6.6.2 for details on enabling synchronous access mode.

If nibble mode has been enabled but `st20run` will not connect then the problem could be:

- Nibble mode is enabled in the PC’s BIOS set up but the port type is still set to some other mode in `vppiset.exe`. See section 6.6.2 for details on setting the port type.
- Nibble mode is enabled in `vppiset.exe` but some other mode (usually ECP) is enabled in the PC’s BIOS setup.

The following is a known problem and workaround with certain PCs:

- The driver fails to detect nibble mode on certain models of HP OmniBook. The BIOS allows “PS/2 mode” or “AT mode” to be selected from the BIOS setup utility. “AT mode” should be selected and the driver should be forced into nibble port mode using `vppiset.exe` (see section 6.6.2).

6.6.4 Windows registry

The driver parameters set by `vppiset.exe` are stored as entries in the Windows registry. The entries listed below are all of type DWORD. They can be edited using the Windows' utility `regedit.exe`.

Under Windows 95 the registry entries are located at:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\VxD\vppist
```

Under Windows NT the registry entries are located at:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\vppist
```

Key	Default	Description
PortType	Auto	Specifies the port type to be used by the driver. If this entry is not defined the driver will attempt to determine this automatically. The valid values for this key are: 0 - Auto Detect 1 - Nibble 2 - Byte 3 - EPP
AccessMode	Auto	Specifies whether the port will used in interrupt or polled mode. If this entry is not defined the driver will attempt to determine this automatically. The valid values for this key are: 0 - Auto Detect 1 - Polled Mode 2 - Interrupt Driven Mode 3 - Synchronous Mode
IRQ	Auto	Specifies the IRQ used by the parallel port driver. If this entry is not defined, the driver will attempt to detect the IRQ automatically.
ReadThreshold	512	The maximum number of bytes to be read in a single burst.
WriteThreshold	512	The maximum number of bytes to be written in a single burst.
PollDelay	1	The minimum number of milliseconds which have to elapse before the kernel may re-schedule the polling routine.

Table 6.2 Registry entries

6.6.5 Connecting to a parallel port

A hardware target must be of type `tap`. This signifies a ST20 Test Access Port, in this case connected to the host through a ST20-JPI interface. The ST20-JPI is connected to a parallel port of a PC host, as in Figure 6.4.

The interface argument for this type of interface is of the form:

`"jpi_ppi` *parallel_port_device_name*`"`

where `jpi_ppi` signifies that the ST20-JPI is directly connected to a parallel port on the PC.

For example, if the target is a ST20 connected via a ST20-JPI to the parallel port LPT1:, then the `target` command is of the form:

`target` *name* `tap "jpi_ppi lpt1"` *config_proc*

where *name* is the name being defined for the target and *config_proc* is the command language target configuration procedure defining the target.

6.7 Defining a simulator target

A simulator target must be of type `st20sim`. The interface argument is a command line to run the simulator, including options. The possible interface options are those listed in Table 6.3 plus any appropriate `st20sim` options, as listed in the "*Using the ST20 simulator tool*" chapter of the "*ST20 Embedded Toolset Reference Manual*".

Option	Description
`-f` *command_file*	Execute the *command_file*, which may contain `processor` and `memory` commands.
`-p` *command_proc*	Execute the procedure *command_proc*, which may contain `processor` and `memory` commands.
`-quiet`	Do not display the version and copyright messages. Also suppress some run-time diagnostic messages.

Table 6.3 Simulator interface options

The simulator does not see the procedure given at the end of the `target` command, so the `-p` option is used to give information about the simulated target to the simulator. For example, the following `target` command defines the simulator target `oursim`, with a target configuration defined by the command language procedure `mysim`:

```
target oursim st20sim "st20sim -q -p mysim"  mysim
```

If the command language procedure `mysim` is not defined in a start-up file then the file that defines it will need to be given to `st20sim` using the `-f` option.

```
target oursim st20sim "st20sim -q -f st20hw.cfg -p mysim"  mysim
```

7 st20run

The `st20run` tool is used for tasks connected with executing an application on a target, including initializing the target, loading code, debugging, tracing and profiling. You can control the debugging, tracing and profiling functionality through breakpoints. Code may be executed on target hardware or a ST20 simulator.

The `st20run` tool loads and runs ST20 programs on ST20 targets. While the program is running, the `st20run` tool provides file and input/output services to the program through the debug functions.

You can run the `st20run` tool in batch mode or interactively. In interactive mode, you can use it as a command line debugger. A windowing graphical interface is also provided which allows you to monitor what is happening while controlling the `st20run` facilities with a mouse through buttons, menus and commands.

The debugger graphical interface is described in Chapter 8. Defining a target is described in Chapter 5. Interfacing to a target is described in Chapter 6.

The "*ST20 Embedded Toolset Reference Manual*" includes descriptions of:

- using `st20run` with STLite/OS20;
- the debugging libraries supplied, which you may use in an application to aid debugging;
- profiling;
- tracing program execution;
- the command language.

7.1 Starting st20run

7.1.1 st20run command line

A `st20run` session is started with a command line. To invoke `st20run`, use the following command line:

➤ `st20run` *{options}* *{filename}*

where: *filename* is the pathname of a linked unit (`.lku`) code file to be loaded and executed on the target or a debug information file (`.dbg`).

options is a list of zero or more of the options given in Table 7.1. You may enter options in upper or lower case, in any order, separated by spaces.

If you give no *filename* or *options* then `st20run` will show brief help information.

st20run can be used in one of three modes:

- st20run can be invoked with the graphical interface described in Chapter 8. The graphical interface is selected by using the -gui command line option. st20run will open a graphical interface window.
- st20run can be invoked in command language mode by using the -debug command line option. In this interactive mode, st20run will display a command language prompt (>).
- st20run can be invoked in batch mode by using the -batch command line option or by specifying a linked unit without the -gui or -debug options.

If neither -gui nor -debug nor -batch is specified, then st20run will start in batch mode if a target and linked unit are specified or command language mode if just a target is specified.

If -gui or -debug are specified on the command line with a target and linked unit, then the debugger will set a breakpoint on main and start the application.

When running in batch mode, if the application executes a debugbreak function call then st20run will switch into command language mode.

The default (simulator) target definitions are found in the file stddefs.cfg in the stdcfg subdirectory of the directory specified by the environment variable ST20ROOT.

Option	Description
`-a[rgs] "arguments"`	Pass arguments to main (`int main(int argc, char *argv[])`).
`-b[atch]`	Run in batch mode. See section 7.1.
`-c[ore] [`*cmd_file*`]`	Create a hosted address space. See section 7.2.9.
`-d[ebug]`	Start command language debugging. See section 7.1.
`-define` *var*`[=`*string*`]`	Set variables to initial string values.
`-g[ui]`	Start the graphical interface. See section 7.1.
`-i[nclude]` *cmd_file*	Execute a command language script. *cmd_file* is executed before `-t` or *{filenames}* parameters are processed.
`-l[ibrary]` *directory*	Include *directory* in the search path.
`-log` *log_file*	Direct the log output to *log_file* in addition to sending it to the display screen.
`-nostartup`	Do not execute any start-up command scripts. See section 7.1.2.
`-p[rofile]`	Apply flat profiling to the program in batch mode and output to the display screen once the program has terminated.
`-r[etries]` *number*	If the target is busy retry *number* times before failing, see the `-sleep` option.
`-s[leep]` *time*	*time* is the time in seconds to wait between retries when the target is busy, see the `-retries` option.
`-t[arget]` *target*	Connect to the specified target, as defined in a `target` command. This is required for loading. See Chapter 5.
`-v[erbose]`	Display extra progress messages.

Table 7.1 `st20run` command line options

7.1.2 Start-up scripts

One or more start-up command language scripts may be executed automatically on starting up `st20run`. On starting up, `st20run` will normally execute the scripts listed in section 1.4.1 in Chapter 1 before processing any command line arguments. This behavior may be turned off by the `-nostartup` command line option.

The start-up command files normally `include` the files containing the target commands for the available targets, and the target configuration commands:

```
## st20run start-up file
directory /st20/examples
include target.cfg

## Useful procedures:
...
```

7.1.3 Examples

The following command line will run a program `hello.lku` in batch mode on the target hardware named `tgt`:

```
% st20run -t tgt hello.lku
hello world
hello world
```

The following command line will start debugging the program `hello.lku` on the target `tgt` using the graphical interface:

```
% st20run -t tgt -g hello.lku
```

The following command line will start interactive command language debugging of the program `hello.lku` on the target `tgt`:

```
% st20run -t tgt -d hello.lku
> breakpoint hit at <hello.c 2>    ## breaks at main
....
> quit
```

The following command line will load the core dump script `save.in`, which will load the saved state that was previously created by a `session` command:

```
% st20run -core save.in
```

The following command line will execute the commands in the file `hw.cfg` before loading the linked unit code file `hello.lku` onto target `tgt`:

```
% st20run -i hw.cfg -t tgt hello.lku
```

The following command line will search `/u/mylibs/` for files, as well as the default debugger search path:

```
% st20run -l /u/mylibs/ -i hw.cfg -t tgt hello.lku
```

The following command line will not execute a start-up command script:

```
% st20run -i hw.cfg -t tgt -nostartup hello.lku
```

The following command line will connect to the target `c1sim` and enter interactive debugging with a command language prompt:

```
% st20run -t c1sim -d
```

The following command line will load `hello.lku` onto the default ST20-C1 target:

```
% st20run -t c1 hello.lku
```

The following command line will load `hello.lku` onto the default ST20-C1 target and pass the arguments into the main procedure:

```
% st20run -t c1 hello.lku -a “one”
```

In this example `argv[0]` will be set to “`hello.lku`” and `argv[1]` will be set to “one”. See the “*Implementation details*” chapter of the "*ST20 Embedded Toolset Reference Manual*" for a description of the arguments to `main`.

7.2 Debugging

This section describes the debugging facility provided by `st20run`, which can be used to debug application software. A debugging process runs on the host, while the application may run on a target ST20 or a simulator.

The debugger allows the user to explore the state of the application. The debugger is controlled by a combination of:

- options on the `st20run` command line,
- using the graphical interface interactively and
- using the command language either interactively or using scripts.

The debugger supports symbolic debugging of C and C++ code and assembler code. It provides the following facilities:

- reading and writing to memory and registers,
- a breakpoint facility that allows the program to be stopped when code is about to be executed or selected variables have been written to or read from;
- a single stepping facility that allows a task to be single stepped at the source level or at the assembly code level;
- a stack trace facility;
- the ability to display the values of variables and display memory;
- a simple interpreter to enable structured C or C++ variables to be displayed;
- a programmable command language that allows complex break points to be set and enables debugging scripts to be generated;
- support for multiple connections, multiple programs and multi-tasking applications;
- profiling;
- tracing.

Depending on the run-time operating system used, a facility to find the tasks of a program is also provided.

The user controls, through the debugger, when the application runs and when it stops. Interactive debugging may be performed using target hardware, as shown in Figure 7.1, or using a simulator running on the host.

The target hardware or simulator runs the application (which is to be debugged). The `st20run` process, running on the host, provides user i/o services for the application and provides the debugger's user interface, displaying the state of the program and passing commands to the Diagnostic Controller Unit (DCU) as requested by the user, see section 7.2.4.

The debugger is not synchronous with the application, so the user has full access to the debugger while code is running on the target or simulator.

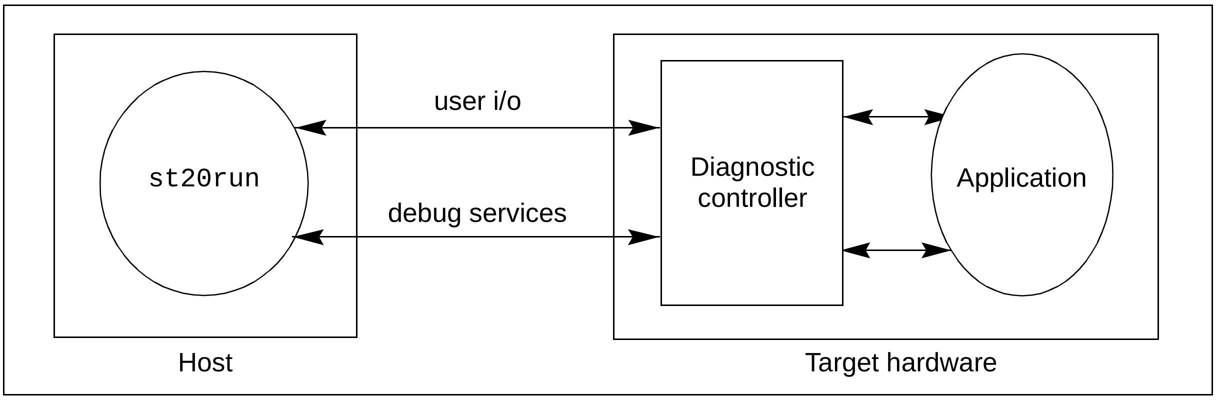


Figure 7.1 Interactive debugging system

7.2.1 Starting debugging

Building the code

In order to use symbolic debugging with ANSI C code, the program should be compiled with the full debugging data option, `-g`, selected on the command line of the build tool `st20cc`. For example:

```
st20cc -g -T appcontrol.cfg -p helloapp hello.c
```

This example compiles and links the source file `hello.c` into a linked unit named `hello.lku`. The name of the output file is derived from the input file and the extension `.lku` is added. `appcontrol.cfg` is a command file which contains a command language procedure called `helloapp` which is specified to describe the target device and to control linking. By default all the symbolic debugging information is put into a debug information file which is generated with the extension `.dbg`.

C code compiled without debugging information is capable of providing a stack trace but cannot provide other symbolic information. In particular, C source code cannot be displayed or referenced.

Code built from assembler code always includes debugging information and does not need the `-g` option when assembling.

Running the debugger

The interactive debugger is started by invoking the `st20run` tool, either with the `-gui` option to use the graphical interface or with the `-debug` option for command line control.

For symbolic debugging, the debugger must have access to the debug files for the application. In GUI or command language mode, the debug files for an application are automatically read-in when a linked unit is specified on the `st20run` command line, or when a linked unit is loaded via the `load` command. Debug files can also be loaded on the `st20run` command line and via the `program -new` command.

GUI session files

A session file may be submitted to the debugger using the `-i` option, in order to initialize the state of the debugging GUI. This file can contain the saved state of a previous debugging session. The session file contains `window`, `program` and `break` commands and is saved via the **Save session** option on the **File** menu of the debugging GUI, see Chapter 8.

If a saved session file is included on the command line via the `-i` option, then further `connect` or `load` commands should not be submitted via the command line.

A session file may have the extension `.ses` and the be included as follows:

`st20run -g -i` *filename*`.ses`

7.2.2 The breakpoint trap handler

On target hardware a breakpoint trap handler has to be installed to enable the debugger to interact with the target and to access the register state of the CPU when a breakpoint occurs. When the trap handler is first required, the debugger searches for a memory segment that has been declared with the attribute `DEBUG.`The debugger loads a breakpoint trap handler into this segment and installs the trap handler, which is then used for breakpoints and target/host interaction. (If necessary a trap handler other than the default can be used by specifying the `traphandler` command after the `chip` command within the configuration file. See Chapter 23 “*Alphabetical list of commands*” in the "*ST20 Embedded Toolset Reference Manual*" for details of the `traphandler` command.

Note: when the breakpoint trap handler is running all interrupts are disabled. Support is included in the breakpoint trap handlers delivered with the toolset to keep an enabled watchdog timer alive while the breakpoint trap handler is running.

7.2.3 The inform mechanism

Input/output functions such as `debugmessage` and `printf` operate via an ‘inform’ mechanism. The host installs a relocatable code unit (RCU) for this inform mechanism in the `DEBUG` segment, following the breakpoint trap handler.

The debugger adds the debug information file ‘`.dbg`’ for this RCU to its program list.

If the inform information is corrupted the debugger reports the error message:

`‘Can’t handle informs’`

If a spurious inform signal is sent to the debugger it is also possible to get the error:

`‘Unidentified target poke; address` *`0xn`*`, data` *`0xn`*`’`

On an inform operation the target signals the host via the DCU and waits in a busy loop until the communication is complete. Interrupts are still enabled whilst the inform operation is running. If the debugger is not connected then the inform operation does not occur. The `informs` command can be used to turn the inform mechanism on and off.

See Chapter 23 “*Alphabetical list of commands*” in the "*ST20 Embedded Toolset Reference Manual*" for details of the `informs` command.

7.2.4 The Diagnostic Control Unit (DCU)

The DCU is coupled to the ST20 CPU, either ST20-C1 or ST20-C2, providing hardware support for debugging target applications.

The toolset provides support for two DCU variants, DCU2 and DCU3. DCU2 has a fixed feature set. DCU3 has a fixed pool of features, but the number and configuration of those features is defined during chip implementation. For example, a DCU3 could be implemented to have between zero and thirty-one compare blocks (compare blocks are used to implement breakpoints). Appendix A describes the hardware breakpoint allocation scheme. For further information describing the DCU hardware, refer to the device data sheet.

The DCU communicates with the outside world via two mechanisms. A simple packet based protocol allows the host/target interface (for example, ST Micro Connect) to communicate with the DCU via the test access port (TAP) interface. Through this mechanism the host can peek and poke target memory and the target can communicate event information and requests for I/O services to the host. The DCU can also be programmed to respond to signals sent via the target’s trigger-in pin and can raise either a pulse or level signal on the target’s trigger-out pin. These can be used with, for example, a Logic State Analyzer (LSA) to coordinate tracing elements of system behavior on a given event or set of events.

The DCU can stop and take control of the target by stalling the CPU or by causing the breakpoint trap handler to run. The stall mechanism is used in two places. On reset, if the debugger is connected to the target, then the CPU is stalled directly prior to executing the chip’s default boot sequence. The stall mechanism is also used when the trace buffer is full and is to be downloaded to the host. The DCU invokes the breakpoint trap handler to stop the user’s application from running when an event such as a hardware breakpoint occurs, or when the user has issued the stop command through the debugger.

For access to CPU registers or, (on an ST20-C2) to peripherals that are mapped into the **PeripheralSpace**, the breakpoint trap handler must be running. All other registers (memory mapped) and memory regions can be accessed via the DCU at any time.

DCU registers are memory mapped into a region of memory that can be peeked and poked through the host/target interface. Whilst CPU registers can only be accessed when the breakpoint trap handler is running and the application is stopped, the DCU provides a register that mirrors the state of the CPU’s **Iptr** register, and a register that encodes information about the CPU’s current status, for example, whether the CPU is busy or idle. These registers can be sampled by `st20run`, allowing profiling that does not affect the application’s real-time behavior.

On receiving an event such as a breakpoint, the debugger can execute command sequences via the `when` command. Through this mechanism complex debug scenarios can be managed. This method of managing a complex debug scenario is not appropriate when the application needs to be run in real-time. This requirement is met by a DCU facility called sequencing. This mechanism allows an event raised by one DCU function to enable or disable another DCU function. This is handled without executing the breakpoint trap handler or stalling the CPU, and therefore without stopping the application. For example, DCU tracing might be enabled on a code breakpoint being hit, or a trigger-in signal being received. Another common scenario is enabling a breakpoint to stop in a commonly executed code section when some other condition is true, say when a count has reached a chosen value.

7.2.5 Examining memory and registers

At any time the memory of the target can be displayed, with the `display` command or GUI Memory Window, without the target having to be stopped. CPU registers can only be displayed when the target is stopped using the `display -r` command or GUI Register Window. On some targets on-chip peripheral registers can only be displayed when the target is stopped. Again, the `display -r` command should be used for this purpose. Similarly if an on-chip device requires non-word aligned access to be made to it, the `display -r` command must be used.

The `register` command can be used to give symbolic names to register addresses which are interpreted by the `display` and `modify` commands. This command can also be used to group sets of registers together for display purposes.

7.2.6 Defining the run-time system

The debugger has several run-time system specific modules that enable it to find tasks and interpret data structures in an RTOS specific way. The `runtime` command is used to define which run-time system the debugger should apply. For example, to make the debugger STLite/OS20 aware:

```
runtime c2os20
```

The `runtime` command is described in Chapter 23 "*Alphabetical list of commands*" in the "*ST20 Embedded Toolset Reference Manual*".

7.2.7 Disconnecting from target hardware

You can end a debugging session by using either `quit` or `disconnect` commands. This does not stop an application running on the target. When `disconnect` is used the debugger closes the current address space, see section 7.2.13. The `quit` command in effect issues a `disconnect` for every address space that is connected.

The address space may alternatively be disconnected by executing the function `debugdisconnect` from within the running application. Again this enables the application to continue running on the target. For applications loaded by the debugger in batch mode, the `debugexit` function will cause the debugger to quit. This function is not supported for interactive use.

7.2.8 Connecting to target hardware that is already booted

To connect to target hardware that is already booted (either from ROM, or from a previous `st20run` invocation), setup a connect procedure that does not reset the target. This connect procedure should use the same `chip` command and memory segment definitions as a resetting connect procedure. It could also load the symbolic debug information of the running program (a `.dbg` file) using the `program -new` command, and if the program produces output to the host screen, the inform mechanism could be enabled using the command `informs -enable`.

Such a connect procedure will look something like this:

```
proc conn_noreset {
   chip STi5512
   memory EXTERNAL ...
   memory ...
   ...
   program -new myprog.dbg
   informs -enable
}

target noreset1 tap "..." conn_noreset
```

If this connect procedure and target definition are in the file `conn.cfg`, the `noreset1` target could be connected to and debugged as follows:

```
% st20run -i conn.cfg -t noreset1 -d
> peek 0x40002000
> ...
> stop
> ...
> go
```

Running an application from ROM is described in Chapter 9

7.2.9 Saving state in a core dump file

Sometimes it may be preferred to save the state of an application in a core dump file. The state may then be explored by reading the dump file without using any target hardware, or the dump file may be reloaded and execution resumed, if the state allows. Saving the state keeps a record for future inspection or reference. Saving the state also frees any target hardware so that, for example, it may be used by other users while the application is being debugged.

The state of the application may be dumped at any time using the `core` command, which produces a hexadecimal dump with extension `.hex`, and a command language script file with extension `.in` to restore the state:

```
st20run -d
> ....                     ## debug session
>core -save coredump       ## create coredump.hex and coredump.in
> ....                     ## resume debugging
```

Subsequent debugging may be performed on a dump file using the `core` command:

```
% st20run -d
> core -load coredump.in ## enter core dump debugging mode
> include coredump.in ## restore debugger state
> ....                ## debug session
```

A dump may be loaded onto target hardware or a simulator by including the core dump script file. After restoring, it may be possible to resume execution, depending on the restored state.

```
% st20run
> connect target
> reset
> go
> ...      ## debug from dump
> quit
%
```

The `-core` option can be used to load a core dump as `st20run` starts.

7.2.10 Code and data breakpoints

A code breakpoint may be set during interactive debugging on a source statement or address range. When a task (that is; a process, or thread of execution) hits the breakpoint it will be stopped immediately before executing that statement or instruction. A statement can have more than one breakpoint set on it.

When an application is multi-tasking, the debugger can treat the code breakpoints of each task as distinct and distinguish between shared code executed by different tasks. For example, a breakpoint may be set in one task so that only that task can hit the breakpoint, while another task executing the same code will not be stopped. Alternatively, a breakpoint may be set so that it stops any task executing that line of code.

A data breakpoint may be set during interactive debugging on a program variable or address range and may watch for read accesses, write accesses or both read and write accesses. If such an access is made to a variable or address range then that task will be stopped. Executing a data breakpoint results in the task being stopped after the access is made.

Code breakpoints are implemented through two mechanisms. Software code breakpoints are implemented by patching a breakpoint instruction into the code at the location of the instruction to be stopped on. The breakpoint instruction causes the breakpoint trap handler to be run. Code breakpoints can also be implemented using the hardware breakpoint facility provided by the DCU.

If the type of breakpoint required is not explicitly specified, then the type of breakpoint setup will be dependent on the memory type at the breakpoint address. For example, if the breakpoint address is a source line in RAM, then a software code breakpoint will be setup, but if the breakpoint address is a source line in ROM then a hardware breakpoint will be setup.

It is unlikely that in normal use the limit on the number of software breakpoints that can be setup will ever be reached, but the number of hardware breakpoints available is limited. Refer to Appendix A for a description of how the debugger allocates hardware breakpoints.

A data breakpoint may be put on an expression, provided that expression defines a block of memory. For example, an array element which has an expression as a subscript may be used.

If a breakpoint address is in a function's stack frame (data breakpoint), the breakpoint is removed when the function returns.

Breakpoints can be set that are local to a particular function or procedure call, so allowing recursive programs to be debugged interactively.

Breakpoints may be set and modified with the `break` command, and then listed, deleted, enabled and disabled via the `events` command. Disabling a breakpoint allows a breakpoint to be temporarily turned off.

Breakpoints may also be set and modified from the graphical interface, as described in Chapter 8.

Hidden breakpoints are set by the debugger while executing `step` and `stepout` commands

When a breakpoint is set, an *event identifier* is returned. This number is output to identify the event whenever the breakpoint is hit or listed.

> `> eventnum = break mux` ## *assign returned event id to* `eventnum`

The `when` command associates a command language script with an event. When the event is hit the script is executed. The `wait` command makes the debugger pause, not executing the next command language command until the application hits an event. The event identifier may be used as a parameter to a `when` or `wait` command in order to wait or act on a specific event.

A one-shot breakpoint may also be set which will automatically be removed after it has been hit for the first time. Counted breakpoints are also supported, which are hit when a variable, statement or address has been accessed a given number of times.

Note:

- A DCU cannot set breakpoints on addresses in an ST20-C2's **PeripheralSpace**.
- A DCU cannot monitor reads to a variable cached in a devices workspace cache.

7.2.11 Stopping the target

A target can be stopped by:

- a `stop` command,
- the application hitting a breakpoint,
- the application executing a user breakpoint.

If the state of the target allows, a stopped target address space can be made to continue by giving the go command. The `interrupts` command can allow the target to restart with interrupts enabled or disabled.

A target address space is stopped by busy waiting in the breakpoint trap handler with interrupts disabled. A simulator address space is stopped by the simulator not executing any more instructions.

When an address space stops, `st20run` tries to find the state of the target when execution stopped, including the values held by the **Iptr** and **Wptr** registers. To do this `st20run` will perform the following steps:

- identify the task that was executing;
- save the task state, which is usually a subset of the target registers;
- symbolically locate to the source code that corresponds to the **Iptr**;
- trace back from the **Wptr** to find all the function calls;
- identify the reason why the execution stopped.

The task state can be modified using the `alter` command. The `go` command will then start the processor executing in the modified task state.

7.2.12 Single stepping

During interactive debugging, a stopped task can be made to execute the next statement, instruction or line of code. This facility is called *single stepping* and is performed using the `step` command. If a task single steps at a function or procedure call, it may optionally step through the function or step over it. *Stepping through* means that the task executes as far as the first statement or instruction within the function. *Stepping over* means that the task executes as far as the return from the function.

The `stepout` command may be used to step out of a function, that is, to continue execution of a task until a specified function returns.

Multiple instructions occurring on the same line may be executed using the `step -line` command.

`step -hardware` causes the `step` command to use a hardware breakpoint to implement the step.

As mentioned in section 7.2.10, stepping is implemented using either software or hardware code breakpoints. Both software and hardware code breakpoint steps can be interrupted through either a scheduling event or an interrupt.

A software step is implemented by patching a software breakpoint instruction into the code at the start of the next statement (or next instruction). The step is completed when the software breakpoint instruction is executed.

Another break event may occur before the step is completed. For example, if a breakpoint is set on a function entry point and the user is stepping through a statement that contains a call to that function. In this case, when the user steps the statement, the function breakpoint is reported. The user must continue from the breakpoint before the step is completed.

In a multi-tasking application, if the code is shared between tasks, other tasks may execute the software breakpoint instruction before the one being stepped. If the debugger is aware of the stack space used for tasks, for example using the STLite/OS20 runtime library, then the step completion will only occur within the original task.

When using the `step` command to step through ROM, the debugger uses a hardware breakpoint. A single address breakpoint is used. That is, a breakpoint is set at the end of the code being stepped. This leads to the same stepping behavior described for a software step.

The command `step -hardware` is implemented through a hardware code breakpoint and will complete for the next **Iptr** that is outside the range of the code being stepped. Therefore if there is a timeslice (and the new **Iptr** is not within the range of the step - a rare scenario, but possible), then the step will complete in the next scheduled task. Or if there is an interrupt, the step will complete in the interrupt handler.

7.2.13 Address spaces

In the context of `st20run`, an address space is a processor state, that is, the contents of memory, the CPU registers and any peripheral registers. The address space may be either:

- the state of a hardware or simulator target, in which case it is called a *target address space*, or
- a loaded core dump or trace dump, in which case it is called a *hosted address space.*

`st20run` allows you to use more than one address space at any one time. If several targets or several core dumps or both are in use simultaneously then each has a separate address space.

A target address space is created by the `connect` command and disconnected by the `disconnect` command. A hosted address space is created with the `core` command. One address space can be copied to another address space using the `copy` command. This allows, for example, a core dump to be loaded onto a target. Both target and hosted address spaces may be removed by using the `space -delete` command.

Two address spaces can be compared using the `compare` command. For example, this could be used to verify that a ROM has been correctly generated by comparing a hosted address space (including the expected ROM image) with the target address space. One address space can be updated with values from another address space using the `copy` command.

Every address space has an identifier. Some commands can take an address space identifier as an argument; if no identifier is given then the current address space is sought from the debugger context. The debugger context is updated when a `connect` or `core` command has occurred or the `context` command has been used.

Each address space must be associated with a *processor type*, defined using a `chip` or `processor` command. This tells `st20run` what registers exist, how to disassemble code and how to format addresses.

Each address space also has one or more *memory segments* associated with it, using the `memory` command. This tells `st20run` which addresses can be read and written and prevents `st20run` from referencing memory addresses that are not defined.

Address spaces can be examined using the `display` and `peek` commands and can be updated for example using the `poke` command. The `peek` and `poke` commands do not check the specified address against known memory segments.

7.2.14 Programs

In the context of `st20run`, the term *program* is used to mean the linked output from `st20cc`, either a linked unit, ROM image file or a relocatable code unit (RCU). For linked units the name of the program is the name of the linked unit, for ROM image files and RCUs the name of the program is the name of the debug information file generated by `st20cc`. The code for a complete application may be split between several programs to allow different sections of code to be loaded independently.

For each debugging session every program is given an integer program identifier, for example 0, 1, 2, 3, ... Some commands can take a program identifier as an argument; if no identifier is given then the program is sought from the debugger context. The debugger context is updated when a `program -new` or `load` command has occurred or the `context` command has been used.

The `program` command can be used to display information about programs. This command can also take a program identifier as a parameter in which case the details of that program are displayed.

The `program` command may also take a program key as a parameter. A '*program key*' is an identifier which is allocated at link time and is unique to that linked output. Usually it is only allocated to ROM image files although it may be allocated to a linked unit if the code specifically references the program key. See the subsections "The *program key*" and "*Setting the value of the program key*" in section 9.3.2 for further details.

The '*current*' program is the program that is executing. The `program` command has several options which operate on the current program.

For full details of the `program` command see the "*ST20 Embedded Toolset Reference Manual*".

7.2.15 Tasks

A task (sometimes called a process or a thread of execution) is a sequential section of code, with associated state, which can be run in parallel with other tasks. A multi-task application is managed by a run-time system, which controls the sharing of CPU time between the tasks and communication between the tasks. The default run-time system uses the special instructions and registers provided by the ST20 to support multi-tasking.

Each task has a task identifier used in the command language. Some commands can take a task identifier as an argument; if no identifier is given then the current task is sought from the debugger context. The debugger context is updated when a breakpoint has occurred or the `context` command has been used.

The `task -update` command/task window can be used to locate all the tasks in an address space. This command can also take a task identifier as a parameter in which case the last known state of that task is displayed.

7.2.16 Examining variables

The value of an expression may be output using the `print` command. A simple C expression syntax is supported to enable the values of structured variables to be displayed.

The debugger can distinguish between instances of automatic variables declared in shared code. An automatic variable is a C variable which is declared inside a function, so multiple executions of the function code give rise to multiple instances of the variable.

Each task that executes the shared code will have a different instance of any variable declared in the shared code. Each instance will be in a different memory location and may have a different value. In this case the user must be careful when inspecting the value of a variable that the correct task is being inspected. A task may be selected by default or explicitly as a command option.

A function that is called recursively may also cause multiple instances of automatic variables. The debugger is able to distinguish these instances by their frames, that is their locations on the stack.

C static variables have only one instance within a program and are not associated with a particular task. If a data breakpoint is set on a C static variable then any task accessing that variable will stop.

Two variables that have the same name need to be distinguished by the debugger. They may have the same name because:

- they are different instances of an automatic variable or
- they have been defined as different local variables in different functions.

Instances of automatic variables can only be distinguished by identifying the frame, as described above.

Two variables defined in different functions or compilation units can be distinguished by their *scope*. The scope of a variable is defined as the section of code where the variable is uniquely defined. In some cases it will be sufficient to specify a program or task; otherwise a statement reference can be used.

7.2.17 Stack trace

Whenever it locates to a point in the code, the debugger attempts to identify the sequence of function calls that led to the current context. It does this by working backwards through the stack to find where each function was called from, and identifying the source for the calling function. This procedure is called backtracing. If the debugger cannot backtrace then it cannot display the full stack trace.

The debugger will stop backtracing when it has backtraced 1000 times. This is used to prevent the debugger going into an endless loop.

Backtracing depends on the debugger being able to find the appropriate debug records. The debugger can backtrace:

- from C code when it has found the debug records, whether or not the `-g` compiler debug option was used.
- from assembly code when it has found the debug records and the appropriate macros have been used on function definition and work space adjust.

The backtrace can be displayed using the `where` command or the call stack window, which gives details of the nested function calls of the task which have not returned. Each function call of a stack trace is called a *frame* and is allocated a number known as the *frame identifier*. The frame identifiers are shown in the stack trace in the first column.

7.3 Commands

`st20run` has a scripting language defined in the "*ST20 Embedded Toolset Reference Manual*". Commands can be entered interactively at the command language prompt, including executing a command language script from a command file. A command file can be used as a batch file from the `st20run` command line using the `-i` option. Commands may also be sent by the application to `st20run`.

The commands recognized by `st20run` are listed in functional groups in the following sections and are identified by “`st20run`” in the tool environment section of each command definition in the "*ST20 Embedded Toolset Reference Manual*".

7.3.1 Hardware configuration

The hardware configuration commands listed in Table 7.2 can be used to define and describe available target hardware and simulators.

Command	Description
`chip`	Define the chip type.
`memory`	Declare a memory segment for the connected target.
`poke`	Write to the memory of the connected target.
`processor`	Define the core type of the processor.

Table 7.2 Hardware configuration commands

7.3.2 Target interface commands

A target, defined by a `target` command, can be connected, described, reset and disconnected using the commands listed in Table 7.3.

Command	Description
`connect`	Connect to a target.
`disconnect`	Close the current target connection.
`reset`	Resets a target board st20run is connected to.
`target`	Define or list hardware or simulator targets.

Table 7.3 Target interface commands

```
> include target.cfg
> target oursim st20sim "st20sim -q -f st20hw.cfg" mysim
> connect oursim
```

Any number of targets can be connected at any one time. A target cannot be manipulated by the debugger until the debugger has connected to it with the `connect` command. The current target is identified by the command language variable `spaceid`, which is set by the `connect` command or by the `context` command, see section 7.3.4.

7.3.3 Boot commands

Programs can be loaded and run on a defined and connected target using the commands listed in Table 7.4.

Command	Description
`bootiptr`	Defines an applications initial **Iptr**.
`fill`	Copy a ROM file into the memory of the connected target.
`load`	Load the specified linked unit file (.lku).
`go`	Start executing on the connected target.
`modify`	Modify register values.

Table 7.4 Boot commands

`load` and `go` can be used to boot from a linked unit file.

```
> connect target1          ## connect to target1
> load hello.lku           ## load the application hello.lku
> go                       ## run the application
```

`fill`, `modify` and `go` can be used to start the processor running a ROM image.

```
> connect c1target
> fill rom1.hex
> modify -r iptr 0x4000000   ## Set initial iptr
> go
```

Note that for ST20-C2 processors the default boot register state, for example **Iptr** and **Wptr** are fixed, but for ST20-C1 systems the reset value of these registers is device implementation dependant.

7.3.4 Debugging commands

The commands in Table 7.5 provide access to the symbolic facilities of the debugger. They enable symbolic information to be loaded into the debugger, and statement mapping and source browsing to be implemented.

Command	Description
`addressof`	Output the address of the specified statement/symbol.
`context`	Display, define and manipulate debugger context information.
`core`	Create a hosted address space.
`entry`	Output the statement reference for the specified symbol.
`informs`	Enables/disables debug input/output.
`modules`	List the modules (source files) of the program.
`program`	Display, define and manipulate program information.
`runtime`	Define the run-time system.
`space`	Output information on address spaces.
`statement`	Output the source statement corresponding to the specified address.
`sizeof`	Output the sizeof of the specified statement/symbol.
`symbols`	List the symbols in the program.
`task`	Output information on known tasks.

Table 7.5 Debugging commands

7.3.5 Execution commands

Table 7.6 lists the commands to halt or continue execution of one or more tasks, and to single step execution of a task.

Command	Description
`interrupts`	Set the global interrupt behavior on restart after an event.
`go`	Start / continue execution.
`restart`	Set the debugger back to its starting state after loading.
`step`	Execute next instruction, statement or line of code.
`stepout`	Continue execution of a function until it returns.
`stop`	Stop execution.

Table 7.6 Execution commands

The application program can be stopped using the `stop` command. This facility may be used if, for example, a task is stuck in a loop.

```
>stop
task 2 main stopped at <hello.c 14>
```

A stopped task may be resumed using the `go` command.

7.3.6 Event commands

Events can be breakpoints, timers or pre-defined events. Special command language variables exist which hold pre-defined event identifiers which are listed in the "*Command language programming*" chapter of the "*ST20 Embedded Toolset Reference Manual*". The user can create breakpoint and timer events and test against all event identifiers.

Each event has a unique integer identifier, which is returned by the command that sets the event. The identifier is used as an argument by subsequent commands operating on the event. The identifier of the last event to be hit is held in the command language variable `eventid`.

The events commands are listed in Table 7.7.

Command	Description
`break`	Set or modify a breakpoint.
`events`	List, delete, enable or disable events.
`remove`	Remove a `when` statement or command language symbol.
`wait`	Wait for an event to be hit.
`when`	Execute *commands* when an event is hit.
`timer`	Create, list, delete, enable or disable a timer event.

Table 7.7 Event commands

An event can be disabled, which means that the event cannot be hit until it has been enabled again.

The command `wait` can be used to explicitly wait for an event to be hit before executing the next command. A `when` statement can be used to associate a command language script with an event hit.

7.3.7 Profiling

Profiling is controlled by the `profile` command or the `-p` command line option. Profiling by either method is described in the "*ST20 Embedded Toolset Reference Manual*".

7.3.8 Task State Commands

When a task is stopped, its state can be examined and altered by the commands in Table 7.8.

Command	Description
`alter`	Write to a register for the defined task.
`print`	Interpret a C expression.
`program`	Display, define and manipulate program information.
`space`	Output information on address spaces.
`task`	Output information on known tasks.
`where`	Generate a stack trace (that is. a function call trace).

Table 7.8 Task state commands

7.3.9 Window management

The graphical interface windows can be managed using the `window` command. This command can open and close windows and set their current and default characteristics. The command has several forms for performing different types of window operations.

7.3.10 Tracing

Tracing program execution is described in the "*ST20 Embedded Toolset Reference Manual*".

7.3.11 Low level debugging

When a target is connected, the commands listed in Table 7.9 are available to inspect and modify the contents of the memory space and registers of the target. These commands can be used for testing the target hardware, low-level debugging and to emulate booting from ROM.

Command	Description
`compare`	Compares two address space memories and registers.
`copy`	Copies the source address space memory and registers to the destination address space.
`display`	Display a block of memory or register of the connected target.
`dump`	Save memory to a file.
`fill`	Copy a ROM file into the memory of the connected target.
`memset`	Fill a block of memory with one value.
`modify`	Write to memory on the connected target.
`peek`	Read the memory of the connected target.
`poke`	Write to the memory of the connected target.
`register`	Associate symbolic name to peripheral register address.

Table 7.9 Low level commands

A memory viewing function allows segments of memory to be viewed in various formats. A low level view of a task can be selected allowing its code to be disassembled, its workspace to be examined and allowing the use of breakpoints and single stepping at instruction level.

7.3.12 File handling commands

Table 7.10 lists the commands for handling files and directories.

Command	Description
`cd`	Change the current working directory.
`directory`	Add *pathname* to the search path.
`include`	Execute the commands in the *filename*.
`mv`	Move a file.
`pwd`	Output the pathname of the current working directory.
`rm`	Remove list of files from the file store.
`searchpath`	Add *pathname* to the search path or output the search path.

Table 7.10 File handling commands

The `directory` and `searchpath` commands perform similar actions, but can be used in different circumstances.

The `directory` command can only be used at top level, that is, not in a procedure or loop. It is also understood by other tools, so it can be used in start-up files.

The `searchpath` command is not understood by other tools.

7.3.13 Other commands

Table 7.11 lists the other commands available when running `st20run`.

Command	Description
`addhelp`	Associates a help string with a procedure name.
`clinfo`	Turn on command language trace output.
`clerror`	Raise an error from a command language program.
`clsymbol`	Get information on command language symbols.
`eval`	Execute a command language procedure.
`fclose`	Closes open file.
`feof`	Tests for end-of-file marker.
`fgets`	Reads one line from a file.
`fopen`	Opens a file for reading, writing or appending.
`fputs`	Writes the specified string to a file.
`help`	Output helpful information.
`log`	Copy all input commands and outputs to a file.
`parse`	Executes a parsed string.
`quit`	Exits the debugger.
`rewind`	Moves the file pointer to the start of the file.
`session`	Saves the current debugging session to a file.
`sys`	Execute a hosts system command.
`write`	Print a message to the debugger output.

Table 7.11 Miscellaneous commands

8 Debugger graphical interface

The debugger has a windowing interface, which gives access to the features of the debugger by means of a keyboard, mouse, windows, menus, buttons and dialogue boxes. This chapter describes the windowing interface, the uses of the menu selections and buttons and the meanings of the displays.

The windowing interface supports multiple tasks, programs and address spaces. Tasks, programs and address spaces are explained in Chapter 7. If you are not using these facilities, you can ignore all references to tasks, programs and address spaces.

8.1 Starting the graphical interface

Building code for debugging is described in Chapter 7. The `-gui` option on the `st20run` command line starts the graphical interface. For example, the following command will start the graphical interface with the compiled application code `hello.lku` using the target `tp3`:

```
st20run -gui -t tp3 hello.lku
```

The target may be a hardware target or a simulator. The `st20run` command line is described in Chapter 7.

A saved debugging session, see section 8.3.3 can be included from the command line by:

`st20run -gui -i` *filename*`.ses`

where *filename*`.ses` is the name of the file, in which the session is saved.

8.2 Windows

The menus and buttons of the windowing interface give access to many features of the debugger. Where more complex features are needed, the full functionality of the debugger can be accessed by typing commands in the Command Console Window, see section 8.17 or via the Command Execution Window, see section 8.10. The command language is described in the "*ST20 Embedded Toolset Reference Manual*".

Several different types of window are provided, as listed in Table 8.1 and described in section 8.3 onwards. Each window displays certain information about the state of the application, the state of the target and the state of the debugger.

Each window has menus to control the display and to perform debugger operations. The meaning of each menu item is displayed at the bottom of the window when the item is selected. Every window has a **File** menu, which includes a **Close** option to close the window, except the Code Window which has an **Exit** option to close down the debugging session. At least one Code Window will always exist the final one of which cannot be closed via **Close**.

Pressing the right-mouse button in a window will usually pop-up a context menu containing commonly used menu options. Only menu items that are valid for the current window, selection, or highlight will be in the pop-up menu. These items are a subset of the items available from the window’s menu bar.

8.2.1 Controlling the GUI

Code Windows are the ‘main’ control windows of the graphical debugger. All other windows can be opened from these and are opened with the data context of the Code Window from which they are opened, see section 8.2.5.

8.2.2 Child windows

Symbols, Map, Callstack, Variables, Registers, Memory and Command Execution Windows are child windows of the Code Window from which they are opened. They will take their data contexts from the parent Code Window, and If the parent window is closed then they will close automatically.

8.2.3 Window Types

Window	Display	Menu and button operations
Code Window	Source or instruction code.	Exit `st20run`, open windows, load and save options, control execution and space, set and clear breakpoints, evaluate expressions, set colors.
Symbols Window	Modules and symbols.	Apply a module and locate to a symbol. Set breakpoint. Open Variables and Memory Windows.
Map Window (Microsoft Windows only)	Memory segments, program sections and symbols.	Locate to a symbol. Set breakpoint.
Callstack Window	Stack trace or instruction stack	Locate source in Code Window. Change call stack.
Variables Window	Values of expressions.	Evaluate C expressions. Open Memory Window.
Registers Window	Values of registers.	Save register values. Add and edit registers.
Memory Window	Contents of memory.	Set breakpoints at addresses. Dump and fill memory to or from file. Set and edit memory.
Tasks Window	Tasks and their current status.	Open Code Window. Stop / continue task.
Targets Window	Available targets and current address spaces.	Connect, disconnect or delete an address space. Open Code Window.
Command Console Window	Manual command history. Debugger output	Enter debugger commands.
Command Execution Window	Debugger command and result	Submit, Enable and disable command.
Trace Window	Traced jumps and source code	Create or interpret a trace.
Trace Generation Window		View and setup trace parameters.
Events Window	Events.	Create and delete events and change event characteristics.
Profile Window	Profile results.	Stop and start profiling. Load and save profile results.

Table 8.1 Window types

8.2.4 Display state

For each task of the application, the debugger interrogates the state of the task only when the task hits an event. The displayed state of a stopped task is normally the current state of that task. When the task is running, the displayed state is the state when the task was last stopped, called the *last known state*.

In some cases you can apply an operation to a particular object by selecting the object in the window display before selecting the operation from the menu. Clicking on or wiping over displayed text selects the text that is highlighted and also selects the line it is in. Double clicking on the display will select a word in the display.

Application program output is displayed in the console from which `st20run` was started.

8.2.5 Data Contexts and Windows

Windows and window elements may have a '*data context*'. Data items will be interpreted within that context as appropriate. For example, the Task Window has an address space context and will only list the tasks that are present in that address space.

Data contexts are required to allow a data item to be uniquely identified. The different forms of data context are:

- Address space, see section 7.2.13.
- Task, see section 7.2.15.
- Program, see section 7.2.14.
- Location:
  - high-level: <module-name line-number statement>
  - low-level: an address
- Frame: callstack frame number.

As an example of the effect of data contexts, consider selecting a variable on a line of source code displayed in the Code Window and selecting Print. To uniquely identify the variable the debugger will use the Code Window's address space and program, it will apply a scope based on the currently loaded module name and the line number on which the variable was selected, and it will apply the frame currently selected on the Callstack Window. If no frame is selected then the debugger will use the last known execution frame of the Code Window's task.

A window inherits its data context from the Code Window it is opened from (or the parent Code Window if opened from a different type of window). On the following windows, this initial data context can be amended by selecting the **Change Context** option from the **View** menu:

- Events
- Trace Generation

- Memory
- Command Execution
- Callstack

A standard **Context Chooser** appears displaying the current settings and allowing them to be changed. Any inappropriate fields are grayed out.

The Variables Window is a special case. Each variable added to the window has its own data context (displayed in the **Scope** field for the currently selected variable).

8.3 Code Window

At least one Code Window will always be displayed. When multiple address spaces are used then there will be at least one Code Window per address space. Creating a new address space from the Targets Window will automatically open a Code Window with that space context.

A Code Window can be opened from the **Windows** menu of the Target Window or the Task Window. Another Code Window with the same initial data context can be opened from the Code Window’s **Windows** menu.

The Code Window may display C or C++ source code, assembler source code or processor instructions. Execution of the application can be controlled from this window, including halting, continuing, single stepping, setting breakpoints, and displaying the values of variables, loading programs, trace files and core files.

A code display shows the last known state of the current task, and by default is updated whenever a task hits an event. An **execution marker** is displayed indicating the current stopped position in the code. The code lines are numbered, and clicking on a line number sets or clears a breakpoint.

Clicking on a code line sets this as the currently **selected line** and a **selection marker** is displayed. If this is a valid code line then the line’s address and statement reference will be shown in the prompt display at the base of the Code Window. If the name of an in-scope variable is highlighted then its value will be shown in the prompt display.

The Code Window also has a set of buttons, used to perform the most common step and execution operations; and list boxes to select a context, that is, a program, module and task. Tasks, programs and address spaces are explained in Chapter 7. If you are not using multiple tasks, programs and address spaces, you can ignore these list boxes.

Each Code Window has an associated Symbols Window, Map Window and Callstack Window. Symbols Windows are described in section 8.4. Map Windows are described in section 8.5. Callstack Windows are described in section 8.6.

8.3.1 Buttons

The Code Window buttons are used to perform the common execution operations.

Go Continue execution of all tasks in the window’s address space.

Stop Interrupt execution of all tasks in the window’s address space.

Locate Change the code display back to the last stopped position of the current task. This button is denoted by a forward arrow →.

Step Step the current task. If source code is displayed then execute one C source statement. If the statement is a function call, then execute the first statement of the function. If instructions are displayed then execute one instruction.

Step Over Step the current task, stepping over function calls. If source code is displayed, then one C source statement is executed. If the statement is a function call, then the function is executed until it returns.

Step Out Complete execution of the selected call stack function (on the Callstack Window), that is, execute until the selected function returns. If no callstack frame is selected then step out one frame level.

Step To Allow execution to proceed to the selected code line.

Find Search for a string in the display (with wrap around). Press ENTER after entering the text to perform the search or select a previous string.

>> Search for the next occurrence of the string selected by **Find**.

8.3.2 List boxes

The list boxes are provided to select the context from all the possible programs, modules and tasks.

Program Select one from the list of programs, (linked units). Selecting a program updates the associated Symbols and Map Windows.

Module Displays currently loaded module. Modules can be selected from the Symbols Window (accessible from the right-mouse button).

Task Select one from the list of tasks, that is, threads of execution.

8.3.3 File menu

Open Text Select a text file for display in the Code Window. If this is recognized as a source code module of a loaded program then the Code Window's program context will change to this program.

Load program

Load a program, (a linked unit), into the Code Window's address space.

Load Dbg Load a debugger symbols file (`.dbg` file), see section 3.2.7.

Load Trace, new space

Create a hosted address space; open a Code Window with this space context; load the selected trace file. **Note**: to load a trace file into an existing address space, use the **Load trace** option on the Trace Window.

Include cfg Include the selected configuration file. This will submit all the commands contained in the file to the debugger.

Add Directory

Add the selected directory to the debugger's search path for locating files.

Identify runtime

Identify the runtime system for this address space.

Load core, new space

Create a hosted address space; open a Code Window with this space context; load the selected core `.in` file and go into core dump debugging.

Save core Create a core file from this address space.

Select Log File

Creates a log file to which all debugger input and output can be written.

Logging Toggles logging on/off for debugger input and output to the previously selected file.

Save session Save the current state of the current debugging session. If a file extension is not provided then the default extension `.ses` will be used. The saved session file contains window, program and break commands.

Restore session

Load a saved debugging session. Restoring a session re-displays the windows, reconnects to targets, reloads programs, and restores breakpoints; it does not attempt to run any programs.

Enter Include Change the code display to the source code file named in the selected `#include` statement.

Exit Include Change the code display back to where **Enter Include** was selected.

Close Close this window. This will close all this window's child windows. If this is the only Code Window then this option will be grayed. If this is the last Code Window for a particular address space then this action will prompt to delete the address space.

Exit st20run Exit from the debugger. A dialog will prompt whether to save the session.

Exit: no save Exit the debugger immediately with no prompt to save.

8.3.4 View menu

Go to line Go to specified line number.

Find text Find a text string in the currently loaded file.

Find Procedure

Locate source code to highlighted procedure name.

Locate Exec Line

Locate source code to current execution line.

Instructions Toggle the code display between the source code and disassembled instructions.

8.3.5 Command menu

Go See button bar **Go**.

Stop See button bar **Stop**.

Restart Restart the current address space. Reruns the connect configuration procedure, reloads all programs, recomputes and resets breakpoints.

Change Exec Line

Changes the current execution line to the **selected line** without executing any instructions (modifies the **Iptr**).

Set Breakpoint

Set a breakpoint on current selection line.

Set Break On Task

Set a task specific breakpoint on the current selection line. This stops execution when the window's current task hits the breakpoint.

Set Break On Frame

Set a frame specific breakpoint on the current selection line. This stops execution when it is hit within the window's current frame.

Set Break Trace On

Tracing begins when this breakpoint is hit (execution is not stopped).

Set Break Trace Off

Tracing stopped when this breakpoint is hit (execution is not stopped).

Delete Breakpoint

Delete all breakpoints on the current selection line.

Delete All Delete all breakpoints in the current address space.

Step See button bar **Step**.

Step Line Step a complete source code line, executing each statement.

Step Over See button bar **Step Over**.

Step Out See button bar **Step Out**.

Step To See button bar **Step To**.

Step Instruction

Single step a machine instruction.

Step Minimal Step into functions compiled with minimal debugging.

Step Thru Minimal

Step through functions compiled with minimal debugging.

Display Variable

Add highlighted variable to Variables Window.

Print Variable Print value of highlighted variable on Command Console Window.

Print* Variable Dereference and print value of highlighted variable on Command Console Window.

Print String Print value of highlighted variable on the Command Console Window as a null-terminated string.

8.3.6 Options menu

The update options are mutually exclusive. Different Code Windows can be set to different update options.

Follow Events Update the Code Window whenever any event is hit and switch its selected task to that of the last stopped event. This is the default operation.

Follow This Task

Update the Code Window whenever the selected task hits an event.

Follow Task Window

Switch the Code Window task to reflect the currently selected task in the list on the Task Window. This option is to allow easy examination of the current position of each task, using the Task Window as a selection control.

Fixed Set the Code Window to not update automatically on events. It will only be updated when the user carries out an action, for example: ‘Locates’, or enters a command.

8.3.7 Windows menu

Windows will be opened with the Code Window’s data context as appropriate. If the required window is already displayed it will be brought to the front.

Command Console There is only one Command Console Window.

Targets There is only one Targets Window.

Events One per address space.

Task One per address space.

Trace One per address space.

Trace Generation
One per address space.

Profile One per address space.

Code Multiple windows. Opened with the initial data context of the Code Window.

Variable Multiple windows. These are child windows of the Code Window.

Memory Multiple windows. These are child windows of the Code Window.

Register Multiple windows. These are child windows of the Code Window.

Cmd Execution
Multiple windows. These are child windows of the Code Window.

Callstack One per Code window. This is a child window of the Code Window.

Symbols One per Code window. This is a child window of the Code Window.

Map (Microsoft Windows only) One per Code Window. This is a child window of the Code Window.

8.3.8 Preferences menu

The **Preferences** menu provides options for setting the default appearance of Windows. Using this requires the `HOME` environment variable to point to a directory which may be written to.

Save as Default
Save the current size and position of this Code Window as the default settings for Code Windows.

Save All as Default
Save the current size and position of all windows as the default settings for their type.

Background colour
Select the text background color.

Text colour Select the foreground text color.

Code font (Microsoft Windows only) Select the font for window contents.

Label font (Microsoft Windows only) Select the font for labels and controls.

8.4 Symbols Window

One Symbols Window is associated with each Code Window. It is opened from the **Windows** menu of a Code Window or by pressing the right-mouse button while in a Code Window. It is used to display and select modules (source code files) of the currently selected program, and the associated symbols.

With no module selected, all the symbols of the program are displayed. Selecting a module causes the symbols of the module and the module details to be displayed. The selected module can be applied to the Code Window by clicking on the **Locate module** button or the **Locate Module** option in the **Command** menu. Selecting a symbol causes the details of the symbol to be displayed. Similarly clicking on the **Locate symbol** button or the **Locate symbol** option in the **Command** menu locates the source code display of the associated Code Window to the currently selected symbol.

Typing into the **Find Module** or **Find Symbol** fields automatically scrolls the **Modules** or **Symbols** list to the matching name.

8.4.1 File menu

Close Close the Symbols Window.

8.4.2 Command menu

Locate Module

Apply the selected module to the associated Code Window.

Locate Symbol

Locate the associated Code Window to the selected symbol. This causes the Code Window to load the module containing the symbol and locate the display to the definition of the symbol.

Set Breakpoint

Put an event on the selected symbol. If the symbol is a function then the event is a breakpoint; if the symbol is a variable then the event is a data breakpoint.

8.4.3 Windows menu

Memory Open Memory Window with address of selected symbol.

Variable Add selected symbol to Variables Window.

8.4.4 Preferences menu

The **Preferences** menu provides options for setting the default appearance of Symbols Windows. Using this requires the `HOME` environment variable to point to a directory which may be written to.

Save as Default

Save the current size and position of this Symbols Window as the default settings for Symbols Windows.

8.4.5 Buttons

Locate module

Load and display the selected module in the associated Code Window.

Locate module & Close

Load and display the selected module in the associated Code Window and close the Symbols Window.

Locate symbol

Locate the associated Code Window to the selected symbol.

Locate symbol & Close

Locate the associated Code Window to the selected symbol and close the Symbols Window.

8.5 Map Window

Microsoft Windows only.

One Map Window is associated with each Code Window. It is opened from the **Windows** menu of a Code Window or by pressing the right-mouse button while in a Code Window. It is used to display and select the memory segments and program sections of the currently selected program, and the associated symbols.

With no memory segment or program section selected, all the symbols of the program are displayed. Selecting a memory segment causes only those symbols placed within the memory segment to be displayed and similarly selecting a program section causes only those symbols placed within the program section to be displayed. Selecting a symbol and then clicking on the **Locate symbol** button or the **Locate symbol** option in the **Command** menu locates the source code display of the associated Code Window to the currently selected symbol.

Typing into the **Find** field and then pressing ENTER searches for a symbol which matches the entered text. If no program section is selected then the search is over the currently selected memory region and if no memory region is selected then the search is over all the symbols in the program.

8.5.1 File menu

Close Close the Map Window.

8.5.2 Command menu

Locate symbol

Locate the associated Code Window to the selected symbol. This causes the Code Window to load the module containing the symbol and locate the display to the definition of the symbol.

Set breakpoint

Put an event on the selected symbol. If the symbol is a function then the event is a breakpoint; if the symbol is a variable then the event is a data breakpoint.

8.5.3 Find menu

Find Open a dialog to find a symbol within the currently selected program section, memory region or program (also updates the **Find** field).

Find next Find the next occurrence of a symbol which matches the text in the **Find** field.

8.5.4 Preferences menu

The **Preferences** menu provides options for setting the default appearance of Windows. Using this requires the `HOME` environment variable to point to a directory which may be written to.

Save as Default
: Save the current size and position of this Map Window as the default settings for Symbols Windows.

8.5.5 Buttons

Find next
: Find the next occurrence of a symbol which matches the text in the **Find** field.

Locate symbol
: Locate the associated Code Window to the selected symbol.

8.6 Callstack Window

One Callstack Window is associated with each Code Window. It is opened from the **Windows** menu of a Code Window

When the Code Window is in source code mode, by default this lists the execution callstack for the Code Window's task. Selecting a frame in the list will locate the Code Window display to the corresponding code line if the relevant module is accessible.

If a frame is selected then choosing **Step Out** on the Code Window will step out to that line. Similarly if a frame is selected, choosing a **Display** or **Print** option on the Code Window will use the selected frame context (see section 8.2.5.)

If the Code Window is in instruction mode then by default the Callstack Window will display the current stack contents, with a marker indicating the current **Wptr** position.

8.6.1 File menu

Close Close the Callstack Window.

8.6.2 View menu

Code Window's mode

Callstack Window follows the instruction/source mode setting of the associated Code Window.

Source Code mode

Displays source code callstack (if available).

Instructions mode

Displays machine stack.

Change Context

Displays Context Chooser. Allows selection of 'Follow Code Window' context (the default), or allows selection of a different task.

8.6.3 Command menu

Step Out Step out to selected frame.

Change Frame Change execution point to selected frame without executing code (changes **Wptr**).

8.6.4 Preferences menu

The **Preferences** menu provides options for setting the default appearance of Windows. Using this requires the `HOME` environment variable to point to a directory which may be written to.

Save As Default

Save the current size and position of this Callstack Window as the default settings for Callstack Windows.

8.7 Variables Window

A Variables Window is a child window of a Code Window. It is opened from the **Windows** menu of a Code Window. Alternatively choosing to ‘Display’ a variable’s value on the Code Window will open a Variables Window if an appropriate one does not exist.

The Variables Window is used to monitor the values of variables defined in the application program, or of C expressions using application variables. The values of the expressions in the display can be updated as the application proceeds. The address of the selected variable or expression and its data context is shown in the **Scope** display. The data context of the variable is taken from the parent Code Window at the time of it being added to the Variables Window.

Clicking on an array or structure in the Variables display will toggle between an expanded and shrunk display of the array or structure. Clicking on a value in the Values display will open a dialogue box to change the value.

The expressions are entered in the **Add variable** field; by selecting an expression in the Code Window and selecting the Code Window **Display** option; or by selecting a variable’s symbol on the Symbols Window and selecting the **Variable** option. The values can be displayed in a range of formats. The values shown may be updated when an event occurs or held fixed, depending on the selection in the **Options** menu.

8.7.1 File menu

Close Close the Variables Window.

8.7.2 View menu

The **View** menu controls the display format of the selected variable.

Hexadecimal Display integer values in hex.

ASCII Display integer values in ASCII.

Decimal Display integer values in decimal.

Octal Display integer values in octal.

Binary Display integer values in binary.

Rtos View Treat selected item as an RTOS variable. This is a toggle switch, where the variable is either displayed in RTOS mode or not. By default, variables are displayed in RTOS mode. This option is independent of the Hexadecimal, ASCII, Decimal, Octal and Binary options.

Value Display value of the selected item.

Dereference Dereference the selected item.

String Dereference the selected item as a null-terminated string.

Array Elements
Open a dialog to define how many elements of the array are displayed.

Change Context

Displays Context Chooser. Allows the currently selected variable's program, task, location or frame to be amended.

Delete Delete the selected item from the list of variables.

8.7.3 Options menu

The **Options** menu controls the update mode of the window.

On Event Update the values whenever an event occurs.

Fixed Do not update the values.

8.7.4 Windows menu

Memory Open Memory Window displaying selected variable's address.

8.7.5 Preferences menu

The **Preferences** menu provides options for setting the default appearance of Variables Windows. Using this requires the `HOME` environment variable to point to a directory which may be written to.

Save as Default

Save the current size and position of this Variables Window as the default settings for Variables Windows.

8.8 Registers Window

A Register Window is a child window of a Code Window. It is opened from the **Windows** menu of a Code Window. It displays all the target CPU registers for the its address space. The Registers Window also allows registers to be modified and for registers to be added and deleted.

8.8.1 File menu

Save Open a dialog to select a file and save the contents of the Registers Window to it.

Close Close the Registers Window.

8.8.2 View menu

The **View** menu is used to control the format of the display.

Binary Display register values in ASCII.

Hexadecimal Display register values in hexadecimal.

Octal Display register values in octal.

Signed Decimal
Display register values as signed decimal.

Unsigned Decimal
Display register values as unsigned decimal.

8.8.3 Edit menu

The **Edit** menu is used to modify the values of registers, to add registers and to delete registers.

Edit selected Open a dialog to modify that value of the selected register in the registers display.

Add Open a dialog to add a register to an existing register group or to a new register group.

Delete Delete the selected register.

8.8.4 Preferences menu

The **Preferences** menu provides options for setting the default appearance of Registers Windows. Using this requires the `HOME` environment variable to point to a directory which may be written to.

Save as Default
Save the current size and position of this Registers Window as the default settings for Registers Windows.

8.9 Memory Window

A Memory Window is a child window of a Code Window. It is opened from the **Windows** menu of a Code Window or Symbols Window, or Variables Window. It displays the contents of a section of memory in the parent Code Window’s address space.

The memory data may be interpreted and displayed in a variety of formats, including characters, integers, floating point numbers and disassembled code. A memory location can be selected for setting a breakpoint.

The memory locations displayed is controlled by the **Address** field which can be entered as an address or as a valid symbol name. The size of the area of memory displayed is controlled by the **Lines** and **Columns** options. If enabled resizing the window increases/decreases the number of lines and columns. Buttons allow paging through memory.

8.9.1 Boxes or buttons

Memory display

Address	Displays the start address of the display. Values can be entered as addresses or symbols. Press ENTER after amending the value to update the displayed memory.
Lines	Controls the number of lines displayed. Press ENTER after amending the value to update the displayed memory.
Cols	Controls the number of columns displayed. Press ENTER after amending the value to update the displayed memory.

Scrolling memory

<<	Page back through memory. The size of the “page” is controlled by the **Columns** and **Lines** settings.
>>	Page forward through memory. The size of the “page” is controlled by the **Columns** and **Lines** settings.
>	Step one line forward in memory.
<	Step one line back in memory.

Searching memory

Find	Search for a string in the display (with wrap around). Press ENTER after entering the text to perform the search or select a previous string.
>>	Search for the next occurrence of the string selected by **Find**.

8.9.2 File menu

Save Contents Select a file and save the currently displayed data to it.

Dump Memory

Opens the Memory Dump dialog. Allows entry of a file name, start address, length, and file type. This area of memory will be dumped to the file in a file type format of hexadecimal, binary, or S-record.

Fill Memory Opens the Memory Fill dialog. Allows entry of a file name, start address, and file type. The file contents will be read into memory. The start address can be omitted for hexadecimal and S-record file types, in which case the memory will be filled starting at the original dump address.

Close Close the Memory Window.

8.9.3 View menu

Change Context

Displays **Context Chooser**. Allows program, task, location or frame to be amended. This context is applied to any symbol name entered into the **Address** field.

Hexadecimal Display integers as hexadecimal.

Decimal Display integers as signed decimals.

Unsigned Decimal

Display integers as unsigned decimals.

Ascii Display integers as ASCII characters

Binary Display integers as unsigned binary.

Octal Display integers as unsigned octal.

Char Display the data as 1-byte integers.

Short Display the data as 2-byte integers.

Long Display the data as 4-byte integers.

Float Display the data as 4-byte IEEE floating point numbers.

Double Display the data as 8-byte IEEE floating point numbers.

Instructions Disassemble the data and display it as instruction mnemonics.

Strings Display the data as null-terminated strings.

Statement Display integers as addresses of statement references.

8.9.4 Command menu

The **Command** menu is used to set breakpoints on selected addresses.

Set Breakpoint Set a breakpoint on the selected address.

Set Data Breakpoint

Set a data breakpoint on the selected address.

Locate to source

Locate the associated Code Window to the source code at the selected address in the display.

Edit selection Open a dialog to allow the value at the selected address in the display to be modified.

Set memory Open a dialog to set a memory range with a repeated byte value.

8.9.5 Preferences menu

The **Preferences** menu provides options for setting the default appearance of Memory Windows. Using this requires the `HOME` environment variable to point to a directory which may be written to.

Save as Default

Save the current size and position of this Memory Window as the default settings for Memory Windows.

Follow dimension

If enabled then the number of lines and columns displayed by the Memory Window is determined by the size of the Memory Window.

8.10 Command Execution Window

The Command Execution Window is a child window of a Code Window. It is opened from **Windows** menu of the Code Window.

This window allows a series of command language expressions to be entered. These will be submitted to the debugger each time an event occurs. The output is displayed in this window. The commands can also be submitted manually and can also be disabled and enabled.

8.10.1 Boxes and buttons

Update output

Buttons controlling the output. **Append** appends to the output display, **Replace** replaces it.

Commands A field in which debugger commands are entered.

8.10.2 File menu

Close Close the Command Execution Window.

8.10.3 View menu

Clear command

Clears the **Commands** field.

Clear output Clears output display.

Change Context

Displays **Context Chooser**. Allows program, task, location, address or frame to be amended. This context is applied to any submitted command.

8.10.4 Command menu

Submit Manually submit the debugger commands immediately.

Disable/Enable

Disable or enable the automatic submission of commands when an event occurs.

8.10.5 Preferences menu

Using this requires the `HOME` environment variable to point to a directory which may be written to.

Save As Default

Save the current size and position of this window as the default for Command Execution Windows.

8.11 Tasks Window

One Tasks Window is associated with each address space. It is opened from the **Windows** menu of a Code Window. The Tasks Window lists all the current tasks in the address space and their status. For multi-tasking programs, this gives the facility to navigate between the tasks and monitor their progress.

Selecting a task displays its details and its current **Iptr**, **Wptr**, **Areg**, **Breg** and **Creg** register values.

8.11.1 File menu

Close Close the Tasks Window.

8.11.2 Command menu

Go Continue execution of all tasks in the address space.

Stop Interrupt execution of all tasks in the address space.

Refresh Refresh the window with the status of all tasks.

8.11.3 Windows menu

Code Open or bring to the front a Code Window for the selected task.

8.11.4 Preferences menu

The **Preferences** menu provides options for setting the default appearance of the Tasks Window. Using this requires the `HOME` environment variable to point to a directory which may be written to.

Save as Default
Save the current size and position of the Tasks Window as the default settings.

8.12 Events Window

One Events Window is associated with each address space. It is opened from the **Windows** menu of a Code Window. The Events Window is used for creating and editing events. Each address space may have one Events Window.

8.12.1 Event modification

A event may be modified by selecting it from the list of current events, modifying its attributes and then pressing the **Update** button or **Update Event** in the **Command** menu.

8.12.2 Event Creation

An event may be created by pressing the **Add** button, selecting the type of event required, setting the expression for the event in the **Expression** field, setting other attributes as necessary and pressing the **Update** button. The expression for an event may be the name of a symbol, an address or a code location (of the format <filename line statement>). Event creation may be abandoned by pressing the **Cancel** button.

8.12.3 Buttons

Enable	Enables the selected event.
Disable	Disables the selected event.
Delete	Deletes the selected event.
Add	Initiates creation of a new event.
Update	Updates the selected event or confirms creation of an event.
Cancel	Cancels the creation of an event.
Locate	Locate the associated Code Window to the source code at the selected event.

8.12.4 File menu

Close Close the Events Window.

8.12.5 View menu

Change Context

Displays Context Chooser. Allows program, task, location or frame to be amended. This context is applied to any symbol name entered into the **Expression** field.

8.12.6 Command menu

Enable Event Enables the selected event.

Disable Event Disables the selected event.

Delete Event Deletes the selected event.

Add Event Initiates creation of a new event.

Update Event Updates the selected event or confirms creation of an event.

Cancel Add Cancels the creation of an event.

Locate source Locate the associated Code Window to the source code at the selected event.

Disable global Disable or enable global interrupts on restart from an event.

8.12.7 Preferences menu

The **Preferences** menu provides options for setting the default appearance of the Events Windows. Using this requires the `HOME` environment variable to point to a directory which may be written to.

Save as Default
Save the current size and position of this Events Window as the default settings for all Events Windows.

8.13 Profile Window

The Profile Window is opened from the **Windows** menu of a Code Window. It is used for profiling application code. Profiling is described in the "*ST20 Embedded Toolset Reference Manual*". Each address space may have one Profile Window.

8.13.1 Buttons

Start	Start profiling.
Stop	Stop profiling.

8.13.2 File menu

New	Deletes all profile information in the Profile Results area.
Load profile	Load the a previously saved profile.
Save profile	Save the text in the Profile Results area to a file.
Close	Close the Profile Window.

8.13.3 Options menu

Start	Start profiling.
Stop	Stop profiling.
Period	Open a dialog in which the sampling period (in microseconds) may be viewed and set.
Sample	Enables or disables sample based profiling.
Idle	Enables or disables idle based profiling.
Flat Info	Displays a flat profile.
Call Info	Displays a call graph profile.
Idle Info	Displays idle period information.
Wrap Idle Trace	Enables wrap mode, whereby once the maximum number of idle periods is reached the start of the list is overwritten.
Idle Size	Open a dialog in which the maximum number of idle records may be viewed and set.

8.13.4 Preferences menu

The **Preferences** menu provides options for setting the default appearance of the Profile Windows. Using this requires the `HOME` environment variable to point to a directory which may be written to.

Save as Default

Save the current size and position of this Profile Window as the default settings for all Profile Windows.

8.14 Trace Window

The Trace Window is opened from the **Windows** menu of a Code Window. It is used for viewing and interpreting traces. Tracing is described in the "*ST20 Embedded Toolset Reference Manual*".

Each address space may have one Trace Window. If there are no address spaces, a trace may be loaded into a hosted address space via the **Load Trace: new space** item on the **File** menu of the Code Window.

A list of the jumps in the trace is displayed, with an index. The jump records are arranged in pages of 1000 records. Selecting a jump will load the appropriate C source file if possible and locate the line in the source code display.

The Trace Window is updated when the trace is ended or when the **Refresh** button is clicked.

8.14.1 Browse buttons

Browse buttons are provided to browse through the trace.

Start	Go to the start of the trace.
<<=	Go back to the previous page of trace records.
=>>	Go on to the next page of trace records.
End	Go to the end of the trace.
Go To	Open a dialogue box to go to a trace record.
Prev	Moves the selection to the previous trace record.
Next	Moves the selection to the next trace record.
Refresh	Refreshes the list of jumps.

8.14.2 Toggle buttons

Two toggle buttons are provided.

Statements	Include source code location statements in the display and when saving a trace.
Instructions	Shows instructions in the source code display region.

8.14.3 File menu

New trace	Reset tracing in preparation for a new trace.
Load trace	Load a previously saved trace.
Save trace	Save the current trace.
Load source	Load any ASCII source file.
Close	Close the Trace Window.

8.14.4 Preferences menu

The **Preferences** menu provides options for setting the default appearance of the Trace Windows. Using this requires the `HOME` environment variable to point to a directory which may be written to.

Save as Default

Save the current size and position of this Trace Window as the default settings for all Trace Windows.

8.15 Trace Generation Window

The Trace Generation Window is opened from the **Windows** menu of a Code Window. It is used to view and set the parameters for generating a trace. The concepts and terms are explained in the "*ST20 Embedded Toolset Reference Manual*". The parameters set in the window can be applied by clicking on the **Apply** button, and the window can be closed by clicking on the **Close** button or selecting **Close** in the **File** menu.

8.15.1 Buffer configuration

Size The trace buffer size in words.

Location The trace buffer base address. The base address must be a multiple of its size. It may be a symbol or an address taken from the Symbols Window (see section 8.4).

8.15.2 Recording mode buttons

These affect the information recorded during tracing.

Jump Froms If selected, the 'jump from' addresses will be recorded.

Jump Tos If selected, the 'jump to' addresses will be recorded.

Stall If selected, the target CPU will be stalled if necessary in order to write each jump record to the buffer.

8.15.3 Buffer full action buttons

These select the behavior when the trace buffer becomes full. Only one may be selected.

Wrap on full Wrap around occurs when the buffer is full.

Extract & go The application is suspended and the buffer contents extracted when the buffer is full. The application and tracing then continues.

Stop on full The application is suspended and the buffer contents extracted when the buffer is full. Tracing is then disabled.

8.15.4 Start and stop buttons and status displays

Two buttons are provided for starting and stopping tracing:

Start Now Starts tracing immediately.

Stop Now Stops tracing immediately.

The following fields display start and stop status information:

Commence The details of when the tracing will start.

Halt The details of when the tracing will stop.

Note that tracing can be started or stopped:

- on a breakpoint;
- manually.

Also tracing can be stopped when the trace buffer is full by selecting **Stop on full**. Not all combinations of starting and stopping conditions are permitted; the valid combinations are listed in the “*Tracing program execution*” chapter of the "*ST20 Embedded Toolset Reference Manual*".

8.15.5 File menu

Close Close the Trace Generation Window.

8.15.6 View menu

Change Context

Displays **Context Chooser**. Allows program to be amended. This context is applied to any symbol name entered into the **Location** field.

8.15.7 Preferences menu

The **Preferences** menu provides options for setting the default appearance of the Trace Generation Windows. Using this requires the `HOME` environment variable to point to a directory which may be written to.

Save as Default

Save the current size and position of this Trace Generation Window as the default settings for all Trace Generation Windows.

8.16 Targets Window

There is only one Targets Window. It is opened from the **Windows** menu of a Code Window. It lists all the current address spaces and the available targets. Selecting an address space displays its details.

A selected address space can be Connected, Disconnected or Deleted.

8.16.1 File menu

Load Core Create a hosted address space; open a Code Window with this space context; load the selected core `.in` file and go into core dump debugging mode.

Close Close the Targets Window.

8.16.2 Command menu

Connect Create a new address space and connect to the currently selected target. This will open a Code Window with this address space context.

Disconnect Disconnect the selected address space from its target. The current information in the debugger is preserved, allowing it to be viewed in other windows.

Delete Disconnect the selected address space and delete all the associated debugger data. This closes all windows with this address space context.

Copy Copy the selected address space into another address space selected from a dialog.

Compare Compare the selected address space with another address space selected from a dialog. The comparison output is displayed on the Command Console Window.

Identify runtime

Identify the runtime system of the selected address space to the debugger.

8.16.3 Windows menu

Code Open a new Code Window with the selected address space context.

8.16.4 Preferences menu

The Preferences menu provides options for setting the default appearance of the Targets Window. Using this requires the `HOME` environment variable to point to a directory which may be written to.

Save as Default

Save the current size and position of the Targets Window as the default settings.

8.17 Command Console Window

The Command Console Window is opened automatically when a message is displayed in the Command Console Window (for example, when an event occurs). You can manually type debugger commands in the Command Console Window, or select a previously entered command from the command history. Commands are submitted in the context of the selected address space, program or task. Output from manually typed commands are displayed in the Command Console Window, and also the results of print commands.

List buttons or boxes at the top of the Command Console Window give the currently selected address space, program and task, and allow a different space, program or task to be selected. The space is indicated by the name of the address space, the symbol ‘=>’ followed by the target to which the address space is connected. The program is shown as the name of the linked unit file. The selected space is also shown in the window title.

8.17.1 File menu

The File menu provides options for restarting the application or closing down the debugger.

Close Close the Command Console Window.

8.17.2 Preferences menu

The Preferences menu provides options for setting the default appearance of the Command Console Windows. Using this requires the `HOME` environment variable to point to a directory which may be written to.

Save as Default

Save the current size and position of the Command Console Window as the default settings.

9 ROM systems

This chapter describes advanced topics related to designing ROM systems using the ST20 toolset. It describes the ROM bootstrap, how ROM programs make external calls, STLite/OS20 awareness, compatibility with earlier toolsets and the chapter has particular emphasis on debugging ROM systems. It explains some of the differences between debugging a system which is downloaded to the target from a linked unit and debugging a stand-alone ROM system; it also describes how to debug multiple programs on a ROM system.

A ROM image can be produced as a binary file (without location information), a hexadecimal file or a Motorola S-record file. The example used in this chapter is based on a hexadecimal file.

Details of how to build a ROM image are given in Chapter 2 through to Chapter 5. Details of the command language used by the toolset are given in the "*ST20 Embedded Toolset Reference Manual*". The Toolset Reference Manual also contains a chapter entitled "*Using st20run with STLite/OS20*", which may be helpful when debugging ROM systems.

9.1 ROM system overview

When booting from a host, chip and C/C++ language initialization is performed by `st20run`. When booting from ROM, these tasks must be performed by the ROM code. When `st20cc` generates a ROM image file it adds bootstrap code to set up the processor and the C/C++ language runtime system. This is called automatically on device reset, before calling the program's '`main`' function.

The ROM C-startup bootstrap code performs the following tasks:

1. Set up a temporary workspace for itself in internal memory. If `initstack` had been specified at link time, the **Wptr** will be set to this command's argument, otherwise the **Wptr** will not be modified. See section 9.4.
2. Call `PrePokeLoopCallback`, as described in section 9.4.
3. Perform all pokes encountered during the link phase.
4. Call `PostPokeLoopCallback`, as described in section 9.4.
5. Set up the program's workspace (specified at link time with the `stack` command).
6. Set up the static link (the pointer to all global data, see the "*Implementation details*" chapter of the "*ST20 Embedded Toolset Reference Manual*").
7. Copy the program code from ROM to RAM if required.
8. Initialize data segments (where statics such as 'static int x=3' are contained and zero bss segments (where statics such as 'static int y') are contained.
9. Set up a *setjmp-longjmp* pair for the `romrestart` function, see the "*ST20 Embedded Toolset Reference Manual*".
10. Install default trap handlers for breakpoints (all cores) and errors (ST20-C2 cores only).
11. Call all C++ static class constructors and functions flagged with `#pragma ST_onstartup`.
12. Set up a *setjmp-longjmp* pair for exit.
13. Call the `main` function of the program.

Note: Source for the bootstrap is shipped with the toolset, so it can be modified for programs which require a bespoke bootstrap procedure.

Note: In section 7.2.4 the effect of various debugging commands on the diagnostic control unit is described. It may be helpful to read this section in order to understand the possible states the debugging session can be placed in.

9.2 An example program and target configuration

An example is provided to illustrate the key issues connected with debugging ROM systems. This example utilizes some of the features commonly encountered in set-top box software. The example files can be found in the `romdebug` subdirectory of the `examples` directory. The example program prints out a set of messages while flashing the LED then uses the on-chip watchdog timer to reset itself.

If the target is reset (for example, by a watchdog timer) while `st20run` is connected the target will stall. The target can only be recovered from the stall state by using the `restart` command. The example calls the function `debugcommand` to execute a command language procedure which will perform a restart after a watchdog reset has occurred.

The examples below apply to an STi5500 device on an EVAL-5500 board. They may be run on a ST20-TP3 device on an STB3-EVAL board by replacing instances of `STi5500MB159` with `ST20TP3MB193B` and adding a target definition to the `example.cfg` file (using the existing definition as a basis, `mk_STi5500_MB159_gen` could be replaced with `mk_ST20TP3_MB193B_gen` and `keen` with the name of your target board).

The example can be built and debugged as a linked unit or as a ROM image.

The definition of targets using the mk_<chip_type>_<board_type>_gen command language procedures defines targets using the default board specification procedures which can be found in the `boards` directory. These default board specification procedures are used by both `st20cc` when linking and `st20run` when loading and debugging. Examining these default procedures, we find several commands key to target definition.

- The `chip` command specifies the variant of the ST20, and defines:
  - The processor core type.
  - The system memory, declared to be `RESERVED`.
  - The internal memory.
  - Cache and interrupt management.
  - A watchdog keep-alive trap handler (see section 7.2.2).
- The `memory` command specifies a memory segment, see the '*ST20 Embedded Toolset Command Language Reference Manual - 72-TDS-533*' for full details of this command.

Stack and heap segments and their sizes are defined. When the size of heap is not specified, it is allocated the remainder of the segment it is in.

The system memory initialization procedure, `ST20C2MemoryInit`, clears various system locations but does not set locations `0x80000040` through `0x8000013c` because these are used by the debugger for trap handlers.

An External Memory Interface (EMI) configuration procedure is also called by most default procedures. This performs a series of pokes in order to allow the board to access EXTERNAL memory.

Instructions to build the example both as a linked unit and as a ROM system have been included so that the differences associated with building and debugging the two formats are demonstrated.

9.2.1 Building a linked unit

A linked unit is generated with the command:

```
st20cc example.c led.c watchdog.c -p STi5500MB159 -M example.map -g -O0
-DENABLE_WATCHDOG
```

The command outputs the linked unit file `example.lku`, the debug information file `example.dbg` and the memory map file `example.map`.

The `-DENABLE_WATCHDOG` option defines the `ENABLE_WATCHDOG` symbol to `example.c`, enabling the use of the `watchdog_reset` function. This is an external command language procedure which can be found in `example.cfg`.

9.2.2 Running and debugging the LKU

The linked unit is loaded onto the target, run and debugged using `st20run`:

```
st20run -i example.cfg -t keen example.lku -g
```

The target `keen` (identified in the command line by `-t keen` and defined within `example.cfg`) has a connect procedure which resets the board (`reset`), tells the debugger about the hardware, initializes the system memory and configures the External Memory Interface (EMI).

The debugger is activated by the `st20run` command line option `-g`. On startup the debugger sets a breakpoint on `main` and executes a `go` command so that the GUI appears with the program ‘waiting’ at the breakpoint on main.

By default, breakpoints will be of software type (that is, instructions poked into the code).

9.2.3 Building a ROM image

A ROM image file is generated with the command:

```
st20cc example.c led.c watchdog.c -p STi5500MB159 -M example_ram.map -g
-O0 -DENABLE_WATCHDOG -romimage -o example_ram.hex
```

The `-DENABLE_WATCHDOG` option defines the `ENABLE_WATCHDOG` symbol to `example.c`, enabling the use of the `watchdog_reset` function. This is an external command language procedure which can be found in `example.cfg`.

All pokes encountered during the link phase will be converted to a table in the ROM image and will be executed by the bootstrap code, so as `STi5500MB159` performs EMI configuration, the ROM image will have access to `EXTERNAL` memory when running. The example output files are generated with the name “`example_ram`” because they are executed in RAM.

By default, program code will be moved to RAM by the bootstrap. Such code will be listed as `MVTORAM` in the map file. To execute code from ROM, the following command must be included at link time:

```
place def_code FLASH
```

This can be done by adding `-T example.cfg` to the command line used to create the ROM image, and exchanging the `-p STi500MB159` option with `-p STi5500MB159_rom`.

A debug information file, `example_ram.dbg`, is created. A ROM image file named `example_ram.hex` is created which represents the memory segment `FLASH`.

Use a flash-burn linked unit (see the `flash` subdirectory of the `examples` directory) or a programmer to burn the ROM/FLASH with the ROM image `example_ram.hex`.

9.2.4 Debugging the ROM image

ROM systems may be debugged provided there is a connection to the host from the on-chip diagnostic control unit of the ST20. ROM systems may be debugged from a reset-processor state or the debugger may be connected while the program is running.

Debugging a boot-from-ROM system (reset)

This debugging mode is started with the following command:

```
st20run -i example.cfg -t keen -g
```

The target `keen` (identified in the command line by `-t keen`) has a connect procedure which resets the board, tells the debugger about the hardware, initializes the system memory and configures the EMI.

The user must then tell the debugger about their program's debugging information. This may be done with the command:

```
program -new example_ram.dbg
```

If there is more than one 'C' or 'C++' program on the ROM, a `program -new` command must be performed for each corresponding debugging information `(.dbg)` file.

The user may set initial breakpoints at this stage. If the code is in `MVTORAM`, breakpoints *must* be of hardware type; this is because software breakpoints would be overwritten when code is moved from ROM by the bootstrap.

The user may start the ROM system by invoking the go command.

If the processor is reset (for instance, by a watchdog timeout) or the program must be re-started, the command `restart` should be used. The commands `reset` followed by `go` are not adequate. The command `restart` executes the connect procedure, sets up breakpoints and executes a go command.

It is important that the connect procedure is executed because this will define the core type, the default register values, the cache and interrupt control procedures and perform EMI programming (which is necessary if the trap handler is not located in internal SRAM).

If the debugger is connected, the chip always stalls on reset. Bit 31 of the DCU2 (or bit 0 of the DCU3) control register is set when the CPU is stalled.

Debugging a running system

To make the example in the `romdebug` directory suitable for connecting to as a running system, it should be recompiled and programmed to ROM without the `-DENABLE_WATCHDOG` option. This turns off the calling of the external command language procedure `watchdog_reset`, which is inappropriate in this case. This is because a debugger is not always connected and the procedure uses the `restart` command. The `restart` command will not work when connected to a target whose connect procedure does not perform a `reset` and EMI initialization.

This debugging mode is started with the following command:

```
st20run -i example.cfg -t keen_nr -g
```

The target `keen_nr` (identified in the command line by `-t keen_nr`) has a connect procedure which does not perform a `reset`, `ST20C2MemoryInit`, or configure the EMI - a `reset` would halt the running program and the other operations were already carried out by the ROM image's bootstrap code in order to start the program. However, the connect procedure does specify the memory segments to the debugger.

By default the debugger's input/output will not appear after connecting to a running system. These are enabled with the debugger command `informs -enable` and may be disabled with the command `informs -disable`.

The user must provide the debugger with the program's debug information using the command:

```
program -new example_ram.dbg
```

At this stage the `stop` button may be pressed and breakpoints may be set. If a runtime is in use (for example STLite/OS20), the debugger should be told at this point with the `runtime` command.

If at a later time the processor is reset, the procedures for initializing the system memory and programming the EMI must be called manually before the `restart` command is issued. Otherwise, the issues discussed under the heading "*Debugging a boot-from-ROM system (reset)*" apply.

9.3 Multiple programs

In section 9.2, a simple example program was run from ROM. This section describes the case where multiple programs are run from ROMs.

9.3.1 Calling to another program

An example of calling from one program to another is provided in the `dualboot` subdirectory of the `examples` directory. This example may be built for both ST20-C2 and ST20-C1 cores. The `dualboot-pre_1_8_1` example directory contains the example in a form which may be built with earlier toolsets. This example may be built for ST20-C2 cores only. The examples illustrate the closing down of STLite/OS20 which must be done before calling another program.

Some key points that are relevant for a ST20-C2 core are illustrated and discussed below to emphasize their importance.

The following code assumes the operating system has been terminated at this point.

(See the “*ROM restart functions*” chapter of the "*ST20 Embedded Toolset Reference Manual*" for an example of closing the operating system.)

```
debugsetmutexfn(NULL, NULL) ;
DISABLE_CACHES ;

if (!debugconnected()) {
  __asm {
    ldab 0xffff, 0;  /* disable all traps */
    trapdis;

    ldab 0xffff, 1;
    trapdis;
  }
} else {
  __asm {
    ldab 0xfffe, 0;  /* disable all traps except */
    trapdis;         /* the debug trap */

    ldab 0xfffe, 1;
    trapdis;
  }
}
debugcommand( “runtime c2rtl”, &res ) ;
__asm {
  ld address ;       /* the entrypoint of the main app */
  ldmemstartval;     /* load 0x80000140 in areg */
  gajw;              /* sets the Wptr */
  pop;               /* remove the old Wptr */

  gcall ;            /* call ‘address’ */
}
```

The debug mutex callback function must be reset to NULL before calling the second program.

Caches *must* be disabled so that when code is moved from ROM to RAM in the C start-up bootstrap for the second program, it is written back to RAM not just to the data cache.

If the debugger is connected, the breakpoint-trap bit in the status register must be left set; if the debugger is not connected, it must be cleared. The breakpoint-trap bit is tested by the C-startup sequence of the second program; if clear it will install a default breakpoint trap handler, if set the current trap handler will be left in place.

Important note: incorrectly setting or clearing the breakpoint-trap bit at this stage will cause dual-boot systems to fail.

On ST20-C1 systems, bit 23 of the status register is used to replicate the functionality of the ST20-C2 core's breakpoint-trap bit.

9.3.2 Debugging multiple programs on a ROM

The debug information (`.dbg`) file should be specified with the command `program -new` for each program on the ROM being debugged.

Multiple programs may share a single region of RAM for their data and for their code. This poses a problem for the debugger because it cannot determine which program is using a shared region of memory at a given point in time. As an example, if a breakpoint is hit while the target is executing, the target will return the value of the current **Iptr** to the debugger. The debugger will search its list of programs for source code which corresponds to this address. If more than one program has code or data at this address during the system's life span, there will be a symbolic information conflict between each program.

This is resolved by the notion of a '*current'* program, which is the program currently executing at a given instant in time. The current program is searched first by the debugger when performing symbolic lookups. If a match is not found, any remaining enabled programs are then searched.

The `program` command has a `-current` option to cater for specifying the current program. Programs may also be disabled from symbolic lookup and re-enabled with the `-disable` and `-enable` options. The enabled/disabled state of a program may be displayed with the `-detail` option. The current program may be displayed with the `context` command.

The program key

A 32 bit unique program key is generated by the linker every time it is invoked, and this program key is placed in a global integer in the application program with the name `_ST_ProgramIdentifier`. The C-startup sequence for a ROM image will execute an `ST_onstartup` function which issues the remote debug command, where *x* is the hexadecimal value of `_ST_ProgramIdentifier`:

```
program -current -uid 0xx
```

This command notifies the debugger of the current program and ensures the debugger starts using this newly executing program for symbolic cross-referencing as early on in the program's life span as is practicably possible.

The program key may also be specified with the `program` command options `-enable` and `-disable`, for example:

```
program -enable -uid 0x1234abcd
```

Preventing program notification

The `ST_onstartup` function described above may be omitted from a ROM system by using the following option with `st20cc`:

```
-W1 p_ST_NoProgramNotify=1
```

Setting the value of the program key

The value of the program key may be explicitly specified at link time, instead of being given a unique value by the linker. This is done by providing the following option to `st20cc`, where *x* is the value to be given to the program key:

```
-W1 p_ST_ProgramIdValue=x
```

Disabling programs

It may be useful in some circumstances to disable a program when the memory it uses will be reused by a program which cannot identify itself as the current program. A typical example is an arrangement with a booter program, a main program and one or more relocatable code units (RCUs) loaded by the main program. The booter program and the RCUs share the same RAM. Just before the booter program calls the main program, it issues the following remote command, where *x* is the value of `_ST_ProgramIdentifier`.

```
program -disable -uid 0xx
```

When RCU(s) are loaded later by the main program, there will be no symbolic conflict for the memory shared by the booter and one or more RCUs. It is recommended that booter programs always disable themselves in the above manner to aid future compatibility. If the processor is restarted, the booter program will need to be re-enabled.

Compatibility with earlier toolsets

The debugger will handle ROM images built with ST20 Embedded Toolset version R1.6.2 upwards. However, it may need assistance where multiple programs exist on a ROM, of which some are built with toolsets earlier than R1.8.1. This is because these will not issue a `program -current` command as an `ST_onstartup` function. Assistance is in the form of either modifying current code or interaction with the program call mechanism. Assistance may not be necessary when the code from multiple programs is executed from ROM because all **Iptr** values will be unique.

If the code is modifiable, the best approach is to create an instance of the variable `int _ST_ProgramIdentifier` for each program which is built with any toolset earlier than version R1.8.1, and place a ‘unique’ value in each. Each program should be provided with an `ST_onstartup` function which issues a remote debug command, where *x* is the hexadecimal value of `_ST_ProgramIdentifier`:

```
program -current -uid 0xx
```

Each function should have a startorder value of at least 100000 to be executed before any system `ST_onstartup` functions and to allow for future toolset development (see `startorder`, in Chapter 23 of the "*ST20 Embedded Toolset Reference Manual*"). The booter should disable itself before calling the main program as described above under the heading "*Disabling programs*".

If the code is not modifiable, the user will have to interact during the debug session to provide assistance to the debugger. The debugger should be connected to the ROM system, and all the `.dbg` files specified with the command `program -new`. Before starting the system with go, the following command should be entered, where *n* is the debugger id of the booter program:

```
program -current n
```

A breakpoint should be set on `romload` of the main program and the system started. When this breakpoint is hit, the following command should be entered, where *m* is the debugger id of the main program:

```
program -current m
```

The following command should be entered, where *n* is the debugger id of the booter program:

```
program -disable n
```

9.4 Callbacks before and after the poke loop in romload()

The set of `poke` commands encountered during the link phase of a ROM program is performed at run-time by the function `romload`. There is no such automatic mechanism for configuration scripts which read memory or perform conditional memory operations. To perform this function, one user-supplied function may be called prior to these pokes being carried out, and another user-supplied function may be called after the pokes.

These functions must be defined as:

```
void PrePokeLoopCallback(void)
void PostPokeLoopCallback(void)
```

They should both be defined in a source module called `initfuncs.c` which is compiled to a file called `initfuncs.tco`. The header file `initfuncs.h` must be included in `initfuncs.c`. The object file `initfuncs.tco` may then be linked into the main program to override stub functions provided in the toolset libraries. Note the object file `initfuncs.tco` must *not* be placed in a library; this object file must be specified to `st20cc` at link time. Also note if one of the callback functions is implemented as described above, the other must be implemented in the same file, even if this is merely a stub.

The user supplied functions may not access static data because it has not been set up at this stage in the bootstrap phase. The functions may declare local variables subject to stack allocation (see below). They should not call functions which have yet to be copied from ROM to RAM.

The functions `romload`, `PrePokeLoopCallback` and `PostPokeLoopCallback` use internal memory for their workspace. If the command `initstack` was present during the link phase, the function `romload` will set the contents of the **Wptr** to the argument of this command as one of its first operations, and before the callback functions are called. Otherwise the **Wptr** will not be altered; if the processor was reset it will thus contain 0x80000140 on an ST20-C2 core, or 0x80000000 on an ST20-C1 core. The function `romload` and the default callback functions use seven words above the **Wptr** value.

If the callback functions use any stack or call any functions, the user must allocate local workspace in internal memory. This is done by specifying the top of a negatively growing stack for user and compiler generated local variables.

For example, to allocate 32 words for the user-supplied callbacks, at the start of `romload` the **Wptr** must be set to:

Wptr `at reset + 32*4`
`= 0x80000140 + 32*4`
`= 0x800001C0`

and this may be achieved by using the following command in the link phase:

```
initstack 0x800001C0
```

Note: seven words above this stack are used by `romload` so the **Wptr** must be set at least seven words below the top of the internal memory segment.

The most convenient way of determining the number of stack words required by a `PrePokeLoopCallback` or `PostPokeLoopCallback` function is to use the stack depth analysis feature in `st20cc`.

9.5 STLite/OS20 awareness

The debugger determines STLite/OS20 state by reading symbolically referenced data structures in the target. If the current program is not specified, the following message may be displayed when the remote command `runtime os20` is issued, and the target may crash.

```
Warning: Incompatible OS/20 interface version number found.
```

The remedy to the above problem is to apply one of the techniques described under the heading "*Compatibility with earlier toolsets*" in section 9.3.2. Note the current program must be correctly specified before the `runtime os20` command is issued. If the `st20cc` option `-runtime os20` is used, the `runtime os20` command will be issued remotely from an `ST_onstartup` function, which is why program notification should be carried out by a higher priority `ST_onstartup` function.

Part 3 - STLite/OS20 Real-Time Kernel

10 Introduction to STLite/OS20

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 STLite/OS20 real-time kernel provides comprehensive multi-tasking services: Tasks synchronize their activities and communicate with each other via semaphores and message queues. Real world events are handled via interrupt routines and communicated to tasks using semaphores. Memory allocation for tasks is selectively managed by STLite/OS20 or the user and tasks may be given priorities and are scheduled accordingly. Timer functions are provided to implement time and delay functions.

The STLite/OS20 real-time kernel is common across all ST20 microprocessors, facilitating the portability of code. The kernel is re-implemented for each core, taking advantage of chip-specific features to produce a highly efficient multi-tasking environment for embedded systems developed for the ST20.

The API (Application Programming Interface) defined in this document corresponds to the 2.08 version of STLite/OS20.

10.1 Overview

The STLite/OS20 kernel features:

- A high degree of hardware integration.
- Multi-priority pre-emptive scheduling based on sixteen levels of priority.
- Semaphores.
- Message queues.
- Timers.
- Memory management.
- Interrupt handling.
- Very small memory requirement.
- Context switch time of 6 microseconds or less.
- Common across all ST20 microprocessors.

Each STLite/OS20 service can be used largely independently of any other service and this division into different services is seen in several places:

- each service has its own header file, which defines all the variables, macros, types and functions for that service, see Table 10.1.
- all the symbols defined by a service have the service name as the first component of the name, see below.

Header	Description
`c1timer.h`	ST20-C1 timer plug-in functions
`cache.h`	Cache functions
`callback.h`	Callback functions
`chan.h`	ST20-C2 specific functions
`device.h`	Device information functions
`interrup.h`	Interrupt handling support functions
`kernel.h`	Kernel functions
`message.h`	Message handling functions
`move2d.h`	Two dimensional block move functions (ST20-C2 specific).
`ostime.h`	Timer functions
`partitio.h`	Memory functions
`semaphor.h`	Semaphore functions
`tasks.h`	Task functions

Table 10.1 STLite/OS20 header files

10.1.1 Naming

All the functions in STLite/OS20 follow a common naming scheme. This is:

service_action[_qualifier]

where *service* is the service name, which groups all the functions, and *action* is the operation to be performed. *qualifier* is an optional keyword which is used where there are different styles of operation, for example, most `interrupt_` functions use interrupt levels, however those with a `_number` suffix use interrupt numbers.

10.1.2 How Part 3 - STLite/OS20 Real-Time Kernel is organized

The division of STLite/OS20 functions into services is used throughout this part. Each of the major service types is described separately, using a common layout:

- An overview of the service, and the facilities it provides.
- A list of the macros, types and functions defined by the service header file.

The remaining sections of this introductory chapter describe the main concepts on which STLite/OS20 is founded. It is advisable to read the remainder of this chapter if you are a first time user.

A ‘*Getting started*’ which describes how to start using STLite/OS20 is provided in Chapter 11.

Chapter 12 describes the STLite/OS20 scheduling kernel.

Chapter 13 describes STLite/OS20 memory and partitions.

Chapter 14 describes STLite/OS20 tasks.

Chapter 15 describes STLite/OS20 semaphores.

Chapter 16 describes STLite/OS20 message handling.

Chapter 17 describes support for real-time clocks.

Chapter 18 describes STLite/OS20 interrupt handling.

Chapter 19 describes STLite/OS20 functions for obtaining ST20 device information.

Chapter 20 describes STLite/OS20 support for caches.

Chapter 21 describes a facility for providing timer support for STLite/OS20 when run on an ST20-C1 core.

Chapter 22 describes support for some ST20-C2 specific features such as channel communication, high priority processes and two dimensional block moves.

Related STLite/OS20 material

A detailed description of each of the functions in STLite/OS20 is given in “*Part 4 - STLite/OS20 functions*” of the "*ST20 Embedded Toolset Reference Manual*".

“*Part 1 - Advanced facilities*” of the Toolset Reference Manual also contains information which is pertinent to STLite/OS20 in the chapters:

- “*Using st20run with STLite/OS20”,*
- “*Advanced configuration of STLite/OS20*”,
- “*Building and running relocatable code*”.

10.2 Classes and Objects

STLite/OS20 uses an object oriented style of programming. This will be familiar to many people from C++, however it is useful to understand how this has been applied to STLite/OS20, and how it has been implemented in the C language.

Each of the major services of STLite/OS20 is represented by a class, that is:

- Memory partitions.
- Tasks.
- Semaphores.
- Message queues.
- Channels.

A class is a purely abstract concept, which describes a collection of data items and a list of operations which can be performed on it.

An object represents a concrete instance of a particular class, and so consists of a data structure in memory which describes the current state of the object, together with information which describes how operations which are applied to that object will affect it, and the rest of the system.

For many classes within STLite/OS20, there are different flavors. For example, the semaphore class has FIFO and priority flavors. When a particular object is created, which flavor is required must be specified by using a qualifier on the object creation function, and that is then fixed for the lifetime of that object. All the operations specified by a particular class can be applied to all objects of that class, however, how they will behave may depend on the flavor of that class. So the exact behavior of `semaphore_wait()` will depend on whether it is applied to a FIFO or priority semaphore object.

Once an object has been created, all the data which represents that object is encapsulated within it. Functions are provided to modify or retrieve this data.

> **Warning**: the internal layout of any of the structure should not be referenced directly. This can and does change between implementations and releases, although the size of the structure will not change.

To provide this abstraction within STLite/OS20, using only standard C language features, most functions which operate on an object take the address of the object as their first parameter. This provides a level of type checking at compile time, for example, to ensure that a message queue operation is not applied to a semaphore. The only functions which are applied to an object, and which do not take the address of the object as a first parameter are those where the object in question can be inferred. For example, when an operation can only be applied to the current task, there is no need to specify its address.

10.2.1 Object Lifetime

All objects can be created using one of two functions:

class`_create`
class`_init`

Normally the *class*`_create` version of the call can be used. This will allocate whatever memory is required to store the object, and will return a pointer to the object which can then be used in all subsequent operations on that object.

However, if it is necessary to build a system with no dynamic memory allocation features or to have more control over the memory which is allocated, then the *class*`_init` calls can be used. This leaves memory allocation up to the user, and allowing a completely static system to be created if required. For *class*`_init` calls the user must provide pointers to the data structures, and STLite/OS20 will use these data structures instead of allocating them itself.

When using *class*`_create` calls, the memory for the object structure is normally allocated from the system partition (the one exception to this is that `tdesc_t` structures are allocated from the internal partition). Thus the partitions must be initialized before any *class*`_create` calls are made. Normally this is done automatically as described in Chapter 11. Chapter 13 describes the system and internal partitions in more detail.

The number of objects which can be created is only limited to the available memory, there are no fixed size lists within STLite/OS20's implementation.

When an object is no longer required, it should be deleted by calling the appropriate *class*`_delete` function. If objects are not deleted and memory is reused, then STLite/OS20 and the debugger's knowledge of valid objects will become corrupted. For example, if an object is defined on the stack and initialized using *class*`_init` then it must be deleted before the function returns and the object goes out of scope.

Using the appropriate *class*`_delete` function will have a number of effects:

- The object is removed from any lists within STLite/OS20, and so will no longer appear in the debugger's list of known objects.
- The object is marked as deleted so any future attempts to use it will result in an error.
- If the object was created using *class*`_create` then the memory allocated for the object will be freed back to the appropriate partition.

Note: the objects created using both *class*`_create` and *class*`_init` are deleted using *class*`_delete`.

Once an object has been deleted, it cannot continue to be used. Any attempt to use a deleted object will cause a fatal error to be reported. In addition, if a task is blocked on an object (for example it has performed a `semaphore_wait()`), and the object is then deleted, the task will be rescheduled, but will immediately raise a fatal error.

10.3 Defining memory partitions

Memory blocks are allocated and freed from memory partitions for dynamic memory management. STLite/OS20 supports three different types of memory partition, *heap*, *fixed* and *simple*, as described in Chapter 13. The different styles of memory partition allow trade-offs between execution times and memory utilization.

An important use of memory partitions is for object allocation. When using the *class*`_create_` versions of the library functions to create objects, STLite/OS20 will allocate memory for the object. In this case STLite/OS20 uses two pre-defined memory partitions (system and internal) for its own memory management. These partitions need to be defined before any of the `create_` functions are called. This is normally performed automatically, see Chapter 11.

10.4 Tasks

Tasks are the main elements of the STLite/OS20 multi-tasking facilities. A task describes the behavior of a discrete, separable component of an application, behaving like a separate program, except that it can communicate with other tasks. New tasks may be generated dynamically by any existing task.

Each task has its own data area in memory, including its own stack and the current state of the task. These data areas can be allocated by STLite/OS20 from the system partition or specified by the user. The code, global static data area and heap area are all shared between tasks. Two tasks may use the same code with no penalty. Sharing static data between tasks must be done with care, and is not recommended as a means of communication between tasks without explicit synchronization.

Applications can be broken into any number of tasks provided there is sufficient memory. The overhead for generating and scheduling tasks is small in terms of processor time and memory.

Tasks are described in more detail in Chapter 14.

10.5 Priority

The order in which tasks are run is governed by each tasks *priority*. Normally the task which has the highest priority will be the task which runs. All tasks of lower priority will be prevented from running until the highest priority task deschedules.

In some cases, when there are two or more tasks of the same priority waiting to run, they will each be run for a short period, dividing the use of the CPU between the tasks. This is called *timeslicing*.

A task’s priority is set when the task is created, although it may be changed later. STLite/OS20 provides the user with sixteen levels of priority.

Some members of the ST20 family of micro-cores implement an additional level of priority via hardware *processes*.

STLite/OS20 supports the following system of priority for tasks running on a ST20-C2 processor:

- Tasks are normally run as low priority *processes*, and within this low priority rating may be given a further priority level specified by the user. Low priority tasks of equal priority are timesliced to share the processor time. Low priority tasks only run when there are no high priority processes waiting to run.
- Tasks may be created to run as high priority *processes*, in which case they are never timesliced and will run until they terminate or have to wait for a time or communication before they deschedule themselves. High priority tasks should be kept as short as possible to prevent them from monopolizing system resources. High priority tasks can interrupt low priority tasks that are running.

On an ST20-C1 there is no hardware priority support. STLite/OS20 allows the user to define individual task priorities, and tasks of equal priority will be timesliced. High priority *processes* are not supported on the ST20-C1.

To implement multi-priority scheduling, STLite/OS20 uses a scheduling kernel which needs to be installed and started, before any tasks are created. This is described in Chapter 12. Further details of how priority is implemented is given in section 14.2.

10.6 Semaphores

STLite/OS20 uses semaphores to synchronize multiple tasks. They can be used to ensure mutual exclusion and control access to a shared resource.

Semaphores may also be used for synchronization between interrupt handlers and tasks and to synchronize the activity of low priority tasks with high priority processes.

Semaphores are described in more detail in Chapter 15.

10.7 Message queues

Message queues provide a buffered communication method for tasks and are described in Chapter 16. On the ST20-C2 they should not be used from tasks running as high priority processes and there are some restrictions on their use from interrupt handlers.

10.8 Clocks

STLite/OS20 provides a number of clock functions to read the current time, to pause the execution of a task until a specified time and to time-out an input communication. Chapter 17 provides an overview of how time is handled in STLite/OS20. Time-out related functions are described in Chapter 14, Chapter 15 and Chapter 16.

On the ST20-C2 microprocessor, STLite/OS20 makes use of the device’s two clock registers, one high resolution, the other low resolution. The number of clock ticks is device dependent and is documented in the device datasheet.

The ST20-C1 microprocessor does not have its own clock and so a clock peripheral is required when using STLite/OS20. This may be provided on the ST20 device or on an external device. A number of functions are required, one to initialize the clock and the others to provide the interface between the clock and the STLite/OS20 functions. STLite/OS20 provides some example sources of such functions which the user can modify for their particular device, see Chapter 21 for details.

10.9 Interrupts

A comprehensive set of interrupt handling functions is provided by STLite/OS20 to enable external events to interrupt the current task and to gain control of the CPU. These functions are described in Chapter 18.

10.10 Device ID

Support is provided for obtaining the ID of the current device, see Chapter 19.

10.11 Cache

A number of functions are provided to use the cache support provided on ST20 devices, see Chapter 20.

10.12 Processor specific functions

The STLite/OS20 API has been designed to be consistent across the full range of ST20 processors. However, some processors have additional features which it may be useful to take advantage of. It should be remembered that using these functions may reduce the portability of any programs to other ST20 processors. See Chapter 21 and Chapter 22.

11 Getting Started with STLite/OS20

This chapter describes how to start using STLite/OS20 and write a simple application. The concepts and terminology used in this chapter are introduced in Chapter 10.

11.1 Building for STLite/OS20

Normally using STLite/OS20 can be almost transparent. All that is necessary is to specify to the linker that the STLite/OS20 runtime system is to be used using the `-runtime` option. For example:

```
st20cc -p STi5500MB159 -runtime os20 app.tco -o system.lku
```

This ensures that by the time the user’s `main` function starts executing:

- the STLite/OS20 scheduler has been initialized and started.
- the interrupt controller has been initialized.
- the system and internal partitions have been initialized.
- thread safe versions of `malloc` and `free` have been set up.
- protection has been installed to ensure that when multiple threads call debug functions, device-independent I/O functions or `stdio` functions concurrently, all operations are handled correctly.

(`st20cc` is described in Chapter 3 and the toolset command language is described in the "*ST20 Embedded Toolset Reference Manual*"‘).

11.1.1 How it works

To initialize STLite/OS20 requires some cooperation between the linker configuration files, and the run time start up code:

- By specifying the `-runtime os20` option to the linker, the configuration file `os20lku.cfg` or `os20rom.cfg` is used instead of the normal C runtime files. This replaces a number of the standard library files with STLite/OS20 specific versions.
- Some modules within the STLite/OS20 libraries contain functions which are executed at start time automatically (through the use of the `#pragma ST_onstartup`).
- A number of symbols are defined by the linker in the STLite/OS20 configuration files, and through the use of the `chip` command. This allows the library code to pick up chip specific definitions, for example, the base address of the interrupt level controller and the amount of available internal memory.
- The heap defined in the configuration files is used for the system partition and so memory for objects defined via *class*`_create` functions is allocated from this heap area. `malloc` and `free` are redefined to allocate memory from the system partition.
- The internal partition is defined to be whatever memory is left unused in the `INTERNAL` memory segment.

All the functions which are called at start up time are standard STLite/OS20 functions. So if the start up code is not doing what is required for a particular application, it is simple to replace it with a custom runtime system and pick and choose which libraries to replace from the C or STLite/OS20 runtimes. The chapter entitled “*Advanced configuration of STLite/OS20*” in the "*ST20 Embedded Toolset Reference Manual*", provides details of how the STLite/OS20 kernel may be recompiled or reconfigured to meet specific application needs. Although this should be done with care and may not be suitable for a production system.

Note: that a ST20-C1 timer module is not installed automatically, because this requires knowledge of how any timer peripherals are being used by the application. See Chapter 21 for further details.

11.1.2 Initializing partitions

The two partitions used internally by STLite/OS20, the system and internal partitions, are set up automatically when the `st20cc -runtime os20` linker command line option is used. However, this relies on information which the user must provide in the linker configuration file.

The system partition uses the memory which is reserved using the `heap` command. As `malloc` and `free` have been redefined to operate on the system partition, the two statements:

```
malloc(size);
```

and

```
memory_allocate(system_partition, size);
```

are now equivalent.

Similarily `calloc` is equivalent to `memory_allocate_clear`, `free` is equivalent to `memory_deallocate` and `realloc` is equivalent to `memory_reallocate`.

The internal partition is defined to be whatever memory is left unused in the `INTERNAL` segment. Thus an `INTERNAL` segment must be defined.

This involves the STLite/OS20 configuration files defining a number of global variables which are read by STLite/OS20 at start up. These are defined using the `addressof`, `sizeof` and `sizeused` commands in the configuration file to give details of the unused portion of the `INTERNAL` segment.

11.1.3 Example

The following example shows how to write a simple STLite/OS20 program, in this case a simple terminal emulator, see Figure 11.1. The code is written to run on an STi5500 evaluation board, but can be easily ported to another target. The device datasheet should be referred to for device specific details.

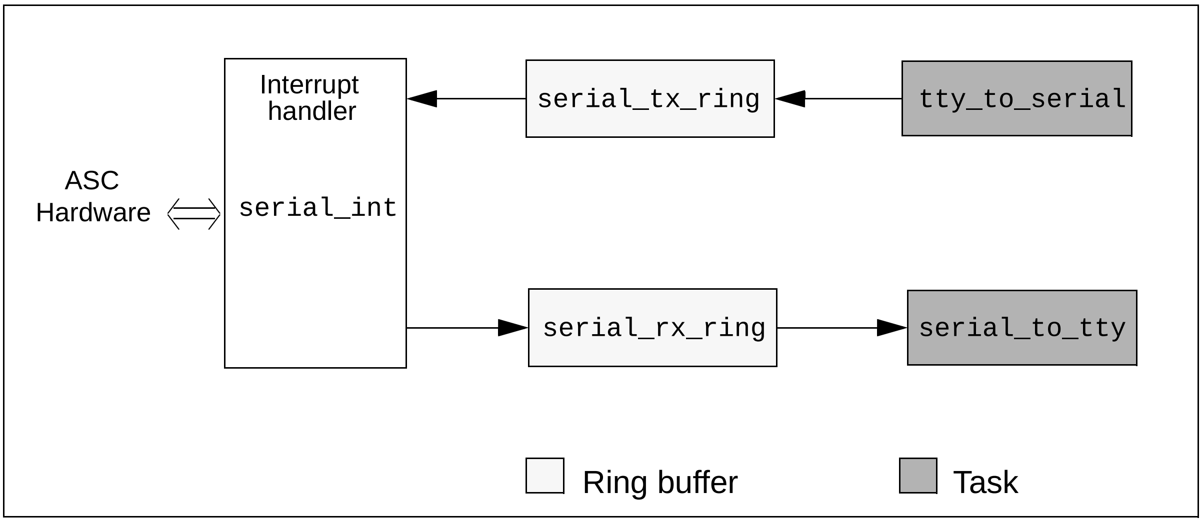


Figure 11.1 Example program schematic

To keep the example concise, some code which does not demonstrate the use of STLite/OS20 is omitted here. The full source code is provided with the STLite/OS20 examples in the `examples/os20/getstart` directory.

The software is structured as two tasks, one handling characters passing from the keyboard and out of the serial port, the other handling characters received from the serial port and being displayed on the console. In addition there is an interrupt handler which services interrupts from the serial hardware.

First some constants and global variables need to be defined:

```
#define CPU_FREQUENCY 40000000
#define BAUD_RATE 9600

#define SERIAL_TASK_STACK_SIZE 1024
#define SERIAL_TASK_PRIORITY 10
#define SERIAL_INT_STACK_SIZE 1024

ring_t serial_rx_ring, serial_tx_ring;
semaphore_t serial_rx_sem;
int serial_mask = ASC_STATUS_RxBUF_FULL;

task_t *serial_tasks[2];
char serial_int_stack[SERIAL_INT_STACK_SIZE];
```

This defines some constants which are needed to initialize the serial port hardware, in particular the CPU frequency, which is needed when programming the serial port hardware’s baud rate generator and may need to be changed when run on another CPU.

It also defines some constants which are needed when setting up the tasks and interrupts, and global variables which are used for communication between the interrupt handler and tasks (the ring buffers and semaphore).

To initialize this system, an initialization function `serial_init` is provided:

```
void serial_init(int loopback)
{
#pragma ST_device(asc)
   volatile asc_t* asc = asc1;

   /* Initialise the PIO pins */
   pio1->pio_pc0_rw = PIO1_PC0_DEFAULT;
   pio1->pio_pc1_rw = PIO1_PC1_DEFAULT;
   pio1->pio_pc2_rw = PIO1_PC2_DEFAULT;

   /* Initial the Rx semaphore */
   semaphore_init_fifo(&serial_rx_sem, 0);

   /* Initialise the ring buffers */
   ring_init(&serial_rx_ring);
   ring_init(&serial_tx_ring);

   /* Install the interrupt handler */
   interrupt_install(ASC1_INT_NUMBER, ASC1_INT_LEVEL,
      serial_int, (void*)asc);
   interrupt_enable(ASC1_INT_LEVEL);

   /* Initialize the serial port hardware */
   asc->asc_baud = CPU_FREQUENCY / (16 * BAUD_RATE);
   asc->asc_control = ASC_CONTROL_DEFAULT |
      (loopback ? ASC_CONTROL_LOOPBACK : 0);
   asc->asc_intenable = serial_mask;

   /* Create the tasks */
   serial_tasks[0] = task_create(serial_to_tty, (void*)asc,
            SERIAL_TASK_STACK_SIZE,
            SERIAL_TASK_PRIORITY, "serial0", 0);
   serial_tasks[1] = task_create(tty_to_serial, (void*)asc,
            SERIAL_TASK_STACK_SIZE,
            SERIAL_TASK_PRIORITY, "serial1", 0);
   if ((serial_tasks[0] == NULL) || (serial_tasks[1] == NULL)) {
      printf("task_create failed\n");
      debugexit(1);
   }
}
```

First the PIO pins need to be set up so that the serial port is connected to the PIO pins (this involves configuring them as ‘alternate mode’ pins, see the device datasheet for details). Next the semaphore used to synchronize the interrupt handler with the receiving task is initialized. Initially this is set to zero to indicate that there are no buffered characters. Each time a character is received, the semaphore will be signalled, in effect keeping a count of the number of buffered characters. This means that the receiving task does not need to check whether the buffer is empty or not when it is run, as long as it waits on the semaphore once per character.

After initializing the ring buffers, the interrupts are initialized. This connects the interrupt handler (`serial_int`) to the interrupt number (`ASC1_INT_NUMBER`). Note that the interrupt level is not configured here. As this may be shared by several interrupt numbers it is good practice to initialize all the levels which are being used in one central location rather than in each module which uses them (see the definition of `main` at the end of this example).

Next the serial port hardware needs to be configured. This sets up the baud rate, enables the port (possibly enabling loopback mode), and enables the interrupts. Initially only receive interrupts are enabled, as there are no characters to transmit yet. However, the handler needs to be notified as soon as a character is received, so receive interrupts are permanently enabled.

Finally the two tasks which will manage the serial communication are created. This will allocate the task's stacks from the system partition, and start them running immediatly.

The next part of the software is the interrupt handler:

```
void serial_int(void* param)
{
   int status;
#pragma ST_device(asc)
   volatile asc_t* asc = (volatile asc_t*)param;

   while ((status = (asc->asc_status & serial_mask)) != 0) {
      switch(status) {
         case ASC_STATUS_RxBUF_FULL:
            ring_write(&serial_rx_ring, asc->asc_rxbuf);
            semaphore_signal(&serial_rx_sem);
            break;
         case ASC_STATUS_TxBUF_EMPTY:
            asc->asc_txbuf = ring_read(&serial_tx_ring);
            if (ring_empty(&serial_tx_ring)) {
               serial_mask &= ~ASC_STATUS_TxBUF_EMPTY;
               asc->asc_intenable = serial_mask;
            }
            break;
      }
   }
}
```

This is constructed as a `while` loop, so that when the loop exits, there are certain to be no interrupts pending[1]. The code needs to be written this way, as the interrupt level is set up to trigger on a rising edge, and so the interrupt must go inactive to guarantee that the next interrupt is seen as a low-to-high transition. An alternative way of constructing this as a high level triggered interrupt is possible, which would cause the interrupt handler to be entered as long as there are pending interrupts.

Inside the loop the code checks for the two cases we are interested in, the receive buffer being full (that is containing a character), and the transmit buffer being empty. Note that the status register is masked by the variable `serial_mask`. This ensures

1. The possibility of error interrupts is ignored in this simple example!

that the code does not check for the transmit buffer being empty when there are no characters to transmit.

The first task takes characters received from the serial port and displays them on the console:

```
void serial_to_tty(void* param)
{
   char c;
   while (running) {
      semaphore_wait(&serial_rx_sem);
      c = ring_read(&serial_rx_ring);
      debugwrite(1, &c, 1);
   }
}
```

This just waits for the semaphore to be signalled, at which point there must be a character in the ring buffer, so this is read and printed.

The second task is slightly more complex. This takes characters typed on the keyboard and sends them to the serial port:

```
void tty_to_serial(void* param)
{
   long int c;
   long int flag;
   const clock_t initial_delay = ONE_SECOND / 100;
   clock_t delay = initial_delay;
#pragma ST_device(asc)
   volatile asc_t* asc = (volatile asc_t*)param;

   while (running) {
      flag = debugpollkey(&c);
      if (flag == 1) {
         interrupt_lock();
         ring_write(&serial_tx_ring, (char)c);
         serial_mask |= ASC_STATUS_TxBUF_EMPTY;
         asc->asc_intenable = serial_mask;
         interrupt_unlock();
      } else {
         task_delay(delay);
         if (delay < (ONE_SECOND / 10)) delay *= 2;
      }
   }
}
```

This code has to poll the keyboard, otherwise while it was waiting for keyboard input it would prevent other tasks doing output. So the code polls the keyboard, and if no character is read, waits for a short while.

If a character is received, then it needs to be written into the transmit ring buffer, and the transmit serial interrupt enabled. This is the only piece of code which needs to be executed with interrupts disabled, as the updating of the ring buffer, `serial_mask` and the serial port’s interrupt enable register needs to be atomic.

Finally a small test harness needs to be provided:

```
int main(int argc, char* argv[])
{
   int loopback = (argc > 1);
   device_id_t    devid = device_id();

   printf("-- Simple Terminal Emulator ---\n");
   printf("OS/20 version %s\n", kernel_version());
   printf("Device %x (%s)\n\n", devid.id, device_name(devid));

   /* Initialise the interrupt system for the chip */
   interrupt_init(ASC1_INT_LEVEL, serial_int_stack,
      sizeof(serial_int_stack),
      interrupt_trigger_mode_rising,
      interrupt_flags_low_priority);
   interrupt_enable(INTERRUPT_GLOBAL_ENABLE);

   serial_init(loopback);

   while (1) {
      debugmessage(".");
      task_delay(ONE_SECOND);
   }
}
```

First this dumps some information to the screen about the STLite/OS20 version and which chip it is running on. Next the interrupt system is initialized, setting up the stack and trigger mode for the interrupt which is going to be used, before enabling global interrupts. The test application is then started, and finally the task goes into an infinite loop dumping a character periodically.

The application can be built as follows:

```
st20cc -p STi5500MB159 example.c -o example.tco -g -c
st20cc -p STi5500MB159 example.tco -o system.lku -runtime os20 -M system.map
```

The first `st20cc` command compiles the source file into a `.tco`. The second command links the application code with the run time libraries, specifying that an STLite/OS20 runtime is to be used.

It can now be run as normal:

```
st20run -t major2 system.lku -args loopback
```

This uses a target called `major2`, and specifies an argument so that the code can be run in loopback mode.

11.2 Starting STLite/OS20 Manually

If the `-runtime` option to `st20link` cannot be used, then it is still possible to use STLite/OS20.

The linker is called with the normal ANSI C runtime libraries and the OS20 libraries are included. This could be achieved, for example, by the following:

```
st20cc -p STi5500MB159 -T myfile.cfg app.tco -o system.lku
```

Where `myfile.cfg` includes the following commands:

```
file os20.lib
file os20intc1.lib
file os20ilc1.lib
```

Note: the two libraries `os20intc1.lib` and `os20ilc1.lib` may need to be replaced by alternative libraries for some devices, see section 18.2 for further details.

STLite/OS20 must then be started and initialized by making the relevant calls from the user code. The order in which initialization and object creation can occur is strictly defined:

1. `partition_init_`*type* can be called to initialize the system and internal partitions. Being able to call this before `kernel_initialize` is a special dispensation for backward compatibility, is not required, and is not encouraged for new programs.
2. `kernel_initialize` should normally be the first STLite/OS20 call made.
3. All class `_init` and `_create` functions can now be called, apart from tasks. This allows objects to be created while guaranteed to still be in single threaded mode.
4. `kernel_start` can now be called to start the multi-tasking kernel. STLite/ OS20 is now fully up and running.
5. Tasks can now be created by calling `task_create` or `task_init`, together with any other STLite/OS20 call.

The one exception to this list is the interrupt system. This has been designed so that it can be used even when the remainder of STLite/OS20 is not being used. Thus calls to `interrupt_init_controller`, `interrupt_init`, `interrupt_install` and any other `interrupt_` function can be made at any point. Obviously any interrupt handlers which run before the kernel has started, should not make calls which can cause tasks to be scheduled, for example `semaphore_signal`.

There is one other piece of initialization which must be performed for the ST20-C1. Before any `time` functions are used, a timer module needs to be installed. For an example of how to do this see Chapter 21.

When STLite/OS20 is used, the heap functions (`malloc`, `calloc`, `free` and `realloc`), debug functions, device-independent I/O functions and stdio functions are thread-safe, see the section “*Concurrency support for libraries*” in the “*Libraries introduction*” in the "*ST20 Embedded Toolset Reference Manual*". **Note**: although thread-safe versions of the heap functions are used they are not mapped to the STLite/OS20 memory management functions as they are when the `-runtime` option is used, see section 11.1.2.

12 Kernel

To implement multi-priority scheduling, STLite/OS20 uses a small scheduling kernel. This is a piece of code which makes scheduling decisions based on the priority of the tasks in the system. It is the kernel's responsibility to ensure that it is always the task which has the highest scheduling priority that is the one which is currently running.

The toolset is supplied with two prebuilt STLite/OS20 kernel libraries: the deployment kernel and debug kernel. The debug kernel is provided to support debugging. Currently the only difference between the two kernels is that the debug kernel has an additional time logging facility. Apart from specific references to the 'debug kernel', the term 'kernel' when used in this chapter applies to either kernel.

12.1 Implementation

The kernel maintains two vitally important pieces of information:

1. Which is the currently executing task, and thus what priority is currently being executed.
2. A list of all the tasks which are currently ready to run. This is actually stored as a number of queues, one for each priority, with the tasks stored in the order in which they will be executed.

The kernel is invoked whenever a scheduling decision has to be made. This is on three possible occasions:

1. When a task is about to be scheduled, the scheduler is called to determine if the new task is of higher priority than the currently executing task. If it is, then the state of the current task is saved, and the new one installed in its place, so that the new task starts to run. This is termed '*preemption*', because the new task has preempted the old one.
2. When a task deschedules, for example it waits on a message queue which does not have any messages available, then the scheduler will be invoked to decide which task to run next. The kernel examines the list of processes which are ready to run, and picks the one with the highest priority.
3. Periodically the scheduler is called to timeslice the currently executing task. If there are other tasks which are of the same priority as the current task, then the state of the current task will be saved onto the back of the current priority queue, and the task at the front of the queue installed in its place. In this way all processes at the same priority get a chance to run.

In this way the kernel ensures that it is always the highest priority task which runs.

The scheduler code is installed as a scheduler trap handler, which causes the ST20 hardware to invoke the scheduling software whenever a scheduling operation is required.

12.2 Time logging

For code running on targets with a ST20-C2 core, STLite/OS20 can be configured to maintain a record of the amount of time each task spends running on the processor. This time logging facility is not available on ST20-C1 cores.

For ST20-C2 cores, the easiest way to perform time logging is to link with the STLite/ OS20 debug kernel. This is done by specifying the `-debug-runtime` option together with the `-runtime os20` option to `st20cc`. For example:

```
st20cc hello.c -debug-runtime -runtime os20 -T sti5500.cfg -p link
```

This facility is introduced under the heading "*Debug and deployment kernels*" in section 3.2.5.

The use of `-debug-runtime` to select the STLite/OS20 debug kernel supersedes an earlier method of enabling time logging by defining `CONF_TIME_LOGGING` and `CONF_INTERRUPT_TIME_LOGGING` in `conf.h` and rebuilding the kernel. This method is still available and is described in the chapter "*Advanced configuration of STLite/OS20*" in the "*ST20 Embedded Toolset Reference Manual*".

12.2.1 Using time logging

The toolset offers the following debugging features in the debug kernel:

- Task time logging maintains a record of the amount of time each task spends running on the processor. The data collected can be accessed using the `task_status` function.
- Interrupt time logging maintains a record of the amount of time in each interrupt and the number of times the interrupt has been called. The functions `interrupt_status` and `interrupt_status_number` are used to return this information.
- System time logging maintains a record of kernel idle time and up-time. The functions `kernel_idle` and `kernel_time` return kernel idle time and kernel up-time respectively. Kernel up-time being the time elapsed since the kernel started.

"*Part 4 - STLite/OS20 functions*" of the "*ST20 Embedded Toolset Reference Manual*" contains descriptions of each of the above functions.

On an ST20-C2, time spent executing high priority processes is not visible to the mechanism used to record idle time. Thus, the time taken to execute a high priority process will be added to the duration of the task or interrupt that was preempted by the high priority process.

Note: all time logging is slightly intrusive. Logging is performed by the target in the scheduler trap and when an interrupt is handled. This could subtly alter the real time performance of the system being logged, however, in most cases the difference in performance should be negligible.

12.3 STLite/OS20 kernel

The primary operations which can be performed on the STLite/OS20 kernel is its installation and start. This is done by calling the functions `kernel_initialize()` and `kernel_start()`. Normally, if the `st20cc -runtime os20` option is specified when linking, this is performed automatically. However, if STLite/OS20 is being started manually the initialization of the STLite/OS20 kernel is usually performed as the first operation in `main()`:

```
if (kernel_initialize () != 0) {
printf (“Error : initialise. kernel_initialize failed\n”);
exit (EXIT_FAILURE);
}
... initialize memory and semaphores ...
if (kernel_start () != 0) {
printf(“Error: initialize. kernel_start failed\n”);
exit(EXIT_FAILURE);
}
```

12.4 Kernel header file: kernel.h

All the definitions related to the kernel are in the single header file, `kernel.h`, see Table 12.1:

Function	Description
`kernel_idle`	Return the kernel idle time.
`kernel_initialize`	Initialize for preemptive scheduling.
`kernel_start`	Starts preemptive scheduling regime.
`kernel_time`	Return the kernel up-time.
`kernel_version`	Return the STLite/OS20 version number.

Table 12.1 Functions defined in `kernel.h`

13 Memory and partitions

Memory management on many embedded systems is vitally important, because available memory is often quite small, and must be used efficiently. For this reason three different styles of memory management have been provided with STLite/OS20, see section 13.2. These give the user flexibility in controlling how memory is allocated, allowing a space/time trade-off to be performed.

13.1 Partitions

The basic job of memory management is to allow the application program to allocate and free blocks of memory from a larger block of memory, which is under the control of a memory allocator. In STLite/OS20 these concepts have been combined into a *partition*, which has three properties:

1. The block of memory for which the partition is responsible.
2. The current state of allocated and free memory.
3. The algorithm to use when allocating and freeing memory.

The method of allocating/deallocating memory is the same whatever style of partition is used, only the algorithm used (and thus the interpretation of the partition data structures) changes.

There is nothing special about the memory which a partition manages. It can be a static or local array, or an absolute address which is known to be free. It can also be a block allocated from another partition, (see the example given in the description of `partition_delete`). This can be useful to avoid having to explicitly free all the blocks allocated:

- Allocate a block from a partition, and create a second partition to manage it.
- Allocate memory from the partition as normal.
- When finished, rather than freeing all the allocated blocks individually, free the whole partition (as a block) back to the partition from which it was first allocated.

The STLite/OS20 system of partitions can also be exploited to build fault-tolerance into an application, by implementing different parts of the application, using different memory partitions. Then if a fault occurs in one part of the application it does not necessarily effect the whole application.

13.2 Allocation strategies

Three types of partition are currently supported in STLite/OS20:

1. *Heap* partitions use the same style of memory allocator as the traditional C runtime `malloc` and `free` functions. Variable sized blocks can be allocated, with the requested size of memory being allocated by `memory_allocate`, and the first available block of memory will be returned to the user. Blocks of memory may be deallocated using `memory_deallocate`, in which case they are returned to the partition for re-use. When blocks are freed, if there is a free block before or after it, it will be combined with that block to allow larger allocations.

   Although the heap style of allocator is very versatile, it does have some disadvantages. It is not deterministic, the time taken to allocate and free memory is variable because it depends upon the previous allocations/ deallocations performed and lists have to be searched. Also the overhead (additional memory which the allocator consumes for its own use) is quite high, with several additional words being required for each allocation.

2. The *fixed* partition overcomes some of these problems, by fixing the size of the block which can be allocated when the partition is created, using `partition_create_fixed` or `partition_init_fixed`. This means that allocating and freeing a block takes constant time (it is deterministic), and there is a very small memory overhead. Thus this partition ignores the `size` argument when an allocation is preformed by `memory_allocate` and uses instead the size argument passed by either `partition_create_fixed` or `partition_init_fixed`.

   Blocks of memory may be deallocated using `memory_deallocate`, in which case they are returned to the partition for re-use.

3. Finally the *simple* partition is a trivial allocator, which just increments a pointer to the next available block of memory. This means that it is impossible to free any memory back to the partition, but there is no wasted memory when performing memory allocations. Thus this partition is ideal for allocating internal memory. Variable sized blocks of memory can be allocated, with the size of block being defined by the argument to `memory_allocate` and the time taken to allocate memory is constant.

The properties of the three partition types are summarized in Table 13.1.

Properties	Heap	Fixed	Simple
Allocation method	As requested by `memory_allocate` or `memory_reallocate`	Fixed at creation by `partition_create_fixed` or `partition_init_fixed`.	As requested by `memory_allocate` or `memory_reallocate`
Deallocation possible	Yes.	Yes.	No.
Overhead size (bytes)	12	4	0
Deterministic	No.	Yes.	Yes.

Table 13.1 Partition properties

13.3 Pre-defined partitions

STLite/OS20 has been designed not to require any dynamic memory allocation itself. This allows the construction of deterministic systems, or for the user to take over all memory allocation.

However, for convenience, all of the object initialization functions (for example, `task_init`, `semaphore_init_fifo`) are also available with a creation style of interface (for example,. `task_create`, `semaphore_create_fifo`), where STLite/OS20 will perform the memory allocation for the object. In these cases STLite/OS20 will use two pre-defined partitions:

- The `system_partition` is used for all object allocation, including semaphores, message queues and the static portion of the task's data structure, including the task's stack. Normally this is managed as a heap partition.
- The `internal_partition` is used just for the allocation of the dynamic part of a task's data structure by `task_create`. To minimize context switch time, this data should be placed in internal memory (see section 14.1 for more information about a task's state). Thus the `internal_partition` should manage a block of memory from the ST20's internal memory. Normally this is managed as a simple partition, to minimize wastage of internal memory.

These partitions *must* be defined before any of the object creation functions are called, and because they are independent of the kernel this can be done before kernel initialization if required.

Normally, if the `st20cc -runtime os20` option is specified when linking, this initialization is performed automatically, see Chapter 11.

If STLite/OS20 is being started manually, the following can be done:

```
partition_t *system_partition;
partition_t *internal_partition;
static int internal_block[200];
static int external_block[100000];
#pragma ST_section(internal_block, “internal_part”)

void initialize_partitions(void)
{
  static partition_t the_system_partition;
  static partition_t the_internal_partition;

  if (partition_init_simple(&the_internal_partition,
   (unsigned char*)internal_block, sizeof(internal_block)) !=0){
      printf(“partition creation failed \n”);
      return;
  }
  if (partition_init_heap(&the_system_partition,
   (unsigned char*)external_block, sizeof(external_block)) !=0){
      printf(“partition creation failed \n”);
      return;
  }

  system_partition = &the_system_partition;
  internal_partition = &the_internal_partition;
}
```

The section `internal_part` is then placed into internal memory by adding a line to the application configuration file:

```
place internal_part INTERNAL
```

13.3.1 Calculating partition sizes

In order to calculate the size of system and internal partitions, several pieces of information are needed for each object created using the `_create` functions (for example, `task_create`, `message_create_queue`, `semaphore_create_fifo`):

- The amount of memory the object requires. Each object is defined by a data structure or type, (refer to individual chapters). For example, `task_t`, refer to 'Chapter 14 *Tasks*'. Table 13.2 lists the different types of object structure that may be created and their memory requirement.

Object structure	Size (words)	Size (Bytes)	Notes
`chan_t`	4	16	ST20-C2 specific.
`semaphore_t`	6	24	
`message_queue_t`	19	76	
`partition_t`	15	60	
`task_t`	9	36	
`tdesc_t`	6 9	24 36	ST20-C2 specific ST20-C1 specific

Table 13.2 Object size requirement

- The amount of memory needed for tasks' stacks, created using `task_create`.
- The amount of overhead the memory allocation function requires for the object. The number of words used depend on whether the object is allocated from a *heap*, *fixed* or *simple* partition. All objects are allocated from the system partition which is managed as a heap partition. The exception is the object structure `tdesc_t` which is allocated from the internal partition (normally managed as a simple partition).Table 13.1 shows the memory overhead associated for each partition type.
- Any additional allocations performed by the user's application.

13.4 Obtaining information about partitions

When memory is dynamically allocated it is important to have knowledge of how much memory is used or how much memory is available in a partition. The status of a partition can be retrieved with a call to the following function:

```
#include <partitio.h>
int partition_status(
    partition_t* Partition,
    partition_status_t* Status,
    partition_status_flags_t flags);
```

The information returned includes the total memory used, the total amount of free memory, the largest block of free memory and whether the partition is in a valid state.

`partition_status()` will return the status of *heap*, *fixed* and *simple* partitions by storing the status into the `partition_status_t` structure which is passed as a pointer to `partition_status()`.

For *fixed* partitions the largest free block of memory will always be the same as the block size of a given *fixed* partition.

13.5 Partition header file: partitio.h

All the definitions related to memory partitions are in the single header file, `partitio.h`, see Table 13.3.

Function	Description
`memory_allocate`	Allocate a block of memory from a partition.
`memory_allocate_clear`	Allocate a block of memory from a partition and clear to zero.
`memory_deallocate`	Free a block of memory back to a partition.
`memory_reallocate`	Reallocate a block of memory from a partition.
`partition_create_simple`	Create a simple partition.
`partition_create_heap`	Create a heap partition.
`partition_create_fixed`	Create a fixed partition.
`partition_delete`	Delete a partition.
`partition_init_simple`	Initialize a simple partition.
`partition_init_heap`	Initialize a heap partition.
`partition_init_fixed`	Initialize a fixed partition.
`partition_status`	Get the status of a partition.

Table 13.3 Functions defined in `partitio.h`

Table 13.4 lists the types defined by `partitio.h`.

Types	Description
`partition_t`	A memory partition.
`partition_stuatus_flags_t`	Additional flags for `partition_status`.

Table 13.4 Types defined by `partitio.h`

14 Tasks

Tasks are separate threads of control, which run independently. A task describes the behavior of a discrete, separable component of an application, behaving like a separate program, except that it can communicate with other tasks. New tasks may be generated dynamically by any existing task.

Applications can be broken into any number of tasks provided there is sufficient memory. When a program starts, there is a single main task in execution. Other tasks can be started as the program executes. These other tasks can be considered to execute independently of the main task, but share the processing capacity of the processor.

14.1 STLite/OS20 tasks

A task consists of a data structure, stack and a section of code. A task's data structure is known as its *state* and its exact content and structure is processor dependent. In STLite/OS20 it is divided into two parts and includes the following elements:

- *dynamic state* defined in the data structure `tdesc_t`, which is used directly by the CPU to execute the process. The fields of this structure vary depending on the processor type. The most important elements of this structure are the machine registers, in particular the instruction (**Iptr**) and workspace (**Wptr**) pointers. A task priority is also used to make scheduling decisions. While the task is running the **Iptr** and **Wptr** are maintained by the CPU, when the task is not executing they are stored in `tdesc_t`. On the ST20-C1 the **Tdesc** register points to the current task's `tdesc_t`.
- *static state* defined in the data structure `task_t`, which is used by STLite/OS20 to describe the task, and which does not usually change while the task is running. It includes the task's state (that is; being created, executing, terminated) and the stack range (used for stack checking).

The dynamic state should be stored in internal memory to minimize context switch time. The state is divided into two in this way so that only the minimum amount of internal memory needs to be used to store `tdesc_t`.

A task is identified by its `task_t` structure and this should always be used when referring to the task. A pointer to the `task_t` structure is called the task's ID, see section 14.12.

The task's data structure may either be allocated by STLite/OS20 or by the user declaring the `tdesc_t` and `task_t` data structures. (These structures are defined in the header file `task.h`). The code for the task to execute is provided by the user function. To create a task, the `tdesc_t` and `task_t` data structures must be allocated and initialized and a stack and function must be associated with them. This is done using the `task_create` or `task_init` functions depending on whether the user wishes to control the allocation of the data structures or not. See section 14.5.

14.2 Implementation of priority and timeslicing

Readers familiar with the ST20 micro-core and STLite/OS20 priority handling may wish to skip to section 14.3 which introduces the facilities provided by STLite/OS20 for influencing priority.

STLite/OS20 implements 16 levels of priority. Tasks are run as the lowest priority hardware *process* for the target hardware with a STLite/OS20 priority specified by the user. STLite/OS20 tasks sit on top of the *processes* implemented by the hardware and use features of the hardware to ensure efficient implementation.

On the ST20-C1, there is no hardware support for multiple priorities.

However on the ST20-C2, the hardware supports two priorities of *processes*, high and low, see Figure 14.1.

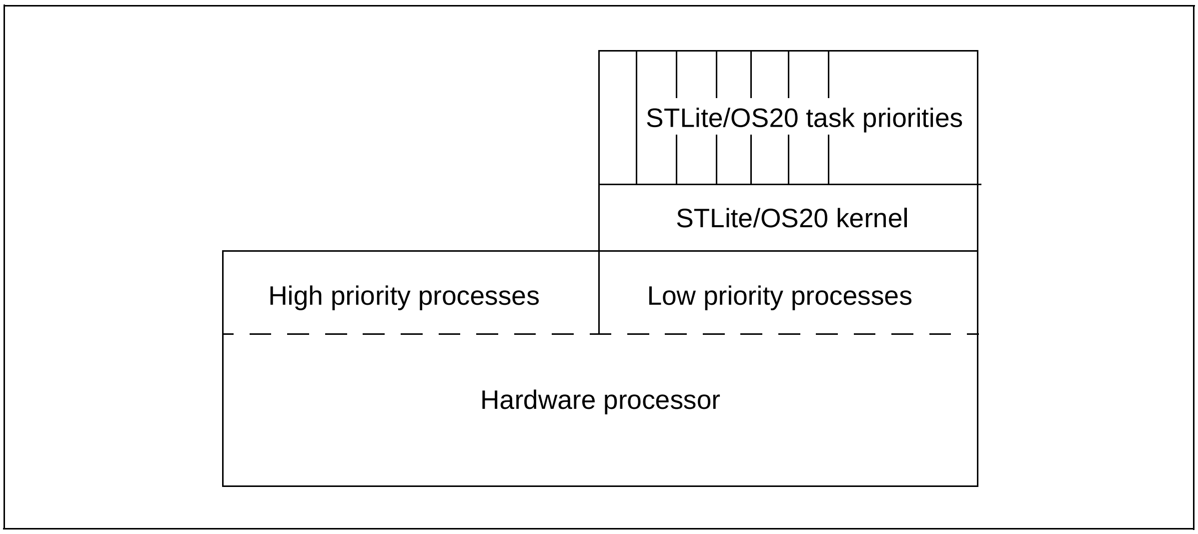


Figure 14.1 ST20-C2 priorities

High priority *processes* take precedence over low priority *processes*, for example, STLite/OS20 tasks. Thus on the ST20-C2, for critical sections of code it is possible to create tasks which use the hardware's high priority *processes* directly.

ST20-C2 high priority *processes* run outside of the STLite/OS20 scheduler, and so some restrictions have to be placed on them:

- They cannot use priority based semaphores
- They cannot use message queues

In addition they inherit two features of the hardware scheduler:

- Tasks are not timesliced, they execute until they voluntarily deschedule
- The units of time are different with high priority *processes* running considerably faster than low priority *processes*. The clock times are device dependent so check the datasheet for actual timings.

14.2.1 Timeslicing on the ST20-C1

On the ST20-C1 microprocessor timeslicing is supported by a *timeslice* instruction. By default timeslicing is disabled by the compiler. However, if the application is compiled with the `st20cc` option `-finl-timeslice` then *timeslice* instructions will be inserted by the compiler. Note, that the runtime libraries are compiled without timeslicing, so it is not possible to timeslice in a library function.

If a *timeslice* instruction is executed when a timeslice is due and timeslicing is enabled then the current *process* will be timesliced, that is, the current *process* is placed on the back of the scheduling queue and the *process* on the front of the scheduling queue is loaded into the CPU for execution.

On some ST20-C1 devices the timeslice clock is provided by peripheral modules and this timeslice clock must be enabled for timeslicing to work. See the device datasheet for details.

Note timeslicing is implemented independently of the clocking peripheral discussed in Chapter 21.

Further details are given in the '*ST20-C1 Core Instruction Set Reference Manual 72-TRN-274*'.

14.2.2 Timeslicing on the ST20-C2

The ST20-C2 microprocessor contains two clock registers. The high priority clock register and the low priority clock register, see Chapter 17.

After a set number of ticks of the high priority clock a timeslice period is said to have ended. When two timeslice period ends have occurred while the same task (low priority hardware process) has been continuously executing, the processor will attempt to deschedule the task. This will occur after the next *j* or *lend* instruction is executed. When this happens the task is descheduled and the next waiting task is scheduled, see Figure 14.2.

High priority processes are never timesliced and will run until completion, or until they have to wait for a communication.

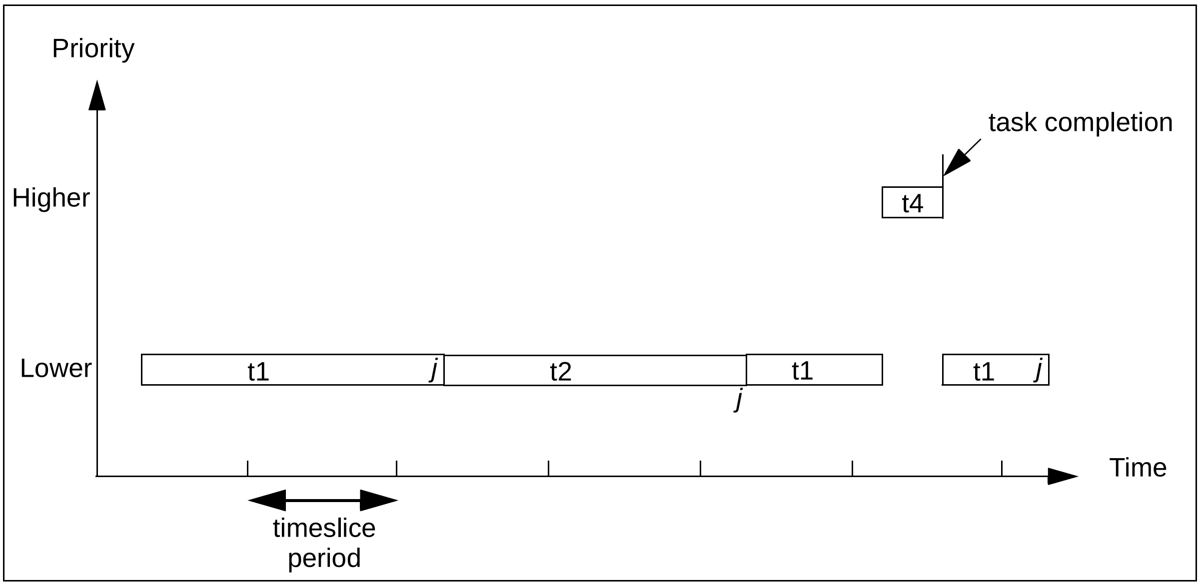


Figure 14.2 Timeslicing on the ST20-C2

A task will nominally run for between one and two timeslice periods. The compiler inserts instructions which allow timeslicing (for example *j*) at suitable points in the code, in order to minimize latency and prevent tasks monopolizing processor time.

If an STLite/OS20 task is preempted by a higher priority STLite/OS20 task then when the lower priority tasks resumes it will start its timeslice period from the beginning of the timeslice period. However, if an STLite/OS20 task is interrupted by an interrupt or preempted by a high priority *process* then it will resume the timeslice period from the point where the interrupt or high priority *process* released the period. Therefore the STLite/OS20 task will loose some of its timeslice.

Further details are given in the '*ST20-C2 Core Instruction Set Reference Manual 72-TRN-273*'.

14.3 STLite/OS20 priorities

The number of STLite/OS20 task priorities and the highest and lowest task priorities are defined using the macros in the header file `task.h`, see section 14.18. Numerically higher priorities preempt lower priorities, for example, 3 is a higher priority than 2.

A task's initial priority is defined when it is created, see section 14.5. The only task which does not have its priority defined in this way is the *root task*, that is, the task which starts STLite/OS20 running by calling `kernel_start`. This task starts running with the highest priority available, `MAX_USER_PRIORITY`.

If a task needs to know the priority it is running at or the priority of another task, it can call the following function:

```
int task_priority (task_t* Task)
```

`task_priority()` retrieves the STLite/OS20 priority of the task specified by `Task` or the priority of the currently active task if `Task` is `NULL`.

The priority of a task can be changed using the `task_priority_set ()` function:

```
int task_priority_set (task_t* Task, int NewPriority);
```

`task_priority_set()` sets the priority of the task specified by `Task`, or of the currently active task if `Task` is `NULL`. If this results in the current task's priority falling below that of another task which is ready to run, or a ready task now has a priority higher than the current task's, then tasks may be rescheduled. This function is only applicable to STLite/OS20 tasks not to high priority hardware processes.

14.4 Scheduling

An active task may either be running or waiting to run. STLite/OS20 ensures that:

- The currently executing task is always the one with the highest priority.

  If a task with a higher priority becomes ready to run then the STLite/OS20 scheduler will save the current task's state and will make the higher priority task the current task. The current task will run to completion unless it is preempted by a higher priority task, and so on. Once a task has completed, the next highest priority task will start executing.

- Tasks of equal priority are timesliced, to ensure that they all get the chance to run. (When compiling for an ST20-C1 a command line option needs to be given, see section 14.2.1).

  Each task of the same priority level will execute in turn for a period of time known as a *timeslice*. See section 14.2.

The kernel scheduler can be prevented from preempting or timeslicing the current task, by using the following pair of functions:

```
void task_lock (void);
void task_unlock (void);
```

These functions should always be called as a pair and can be used to create a critical region where one task is prevented from preempting another. Calls to `task_lock()` can be nested, and the lock will not be released until an equal number of calls to `task_unlock()` have been made. Once `task_unlock ()` is called, the scheduler will start the highest priority task which is available, running. This may not be the task which calls `task_unlock ()`.

If a task voluntarily deschedules, for example, by calling `semaphore_wait` then the critical region will be unlocked and normal scheduling resumes. In this case the subsequent `task_unlock` has no effect. It should still be included in case the task did not deschedule, for example, the semaphore count was already greater than zero.

Note that when this lock is in place the task can still be interrupted by interrupt handlers and high priority processes (on the ST20-C2). Interrupts can be disabled and enabled using the `interrupt_lock ()` and `interrupt_unlock ()` functions, see Chapter 18.

14.5 Creating and running a task

The following functions are provided for creating and starting a task running:

```
#include <task.h>

task_t* task_create (
    void (*Function)(void*),
    void* Param,
    int StackSize,
    int Priority,
    const char* Name,
    task_flags_t flags);
```

```
#include <task.h>

int task_init(
    void (*Function)(void*),
    void* Param,
    void* Stack,
    int StackSize,
    task_t* Task,
    tdesc_t* Tdesc,
    int Priority,
    const char* Name,
    task_flags_t flags);
```

Both functions set up a task and start the task running at the specified function. This is done by initializing the data structures `tdesc_t` and `task_t` and associating a function with them.

Using either `task_create` or `task_init`, the function is passed in as a pointer to the task's entry point. Both functions take a single pointer to be used as the argument to the user function. A cast to `void*` should be performed in order to pass in a single word sized parameter (for example,. an `int`). Otherwise a data structure should be set up.

The functions differ in how the task's data structure is allocated. `task_create` will allocate memory for the task's stack, control block `task_t` and task descriptor `tdesc_t`, whereas `task_init` enables the user to control memory allocation. The task's control block and task descriptor should be declared before the call to `task_init`.

`task_create` and `task_init` both require the stack size to be specified. Stack is used for a function's local variables and parameters, as a guide each function uses:

- Four words for the task to remove itself if it returns.
- Four extra words for the initial user stack.
- On the ST20-C2 six words are needed by the hardware scheduler (for state which is saved into 'negative workspace').
- In some cases the full CPU context needs to be saved on the task's stack. On the ST20-C1 this is always needed when a task is preempted (7 words). On the ST20-C2 it is only needed if a task's priority is changed by another task, or it is suspended (11 words).
- Then recursively:
    - Space for local variables declared in the function, (add up the number of words).
    - Space for calls to extra functions. For a library function allow 150 words for worst case.

For details of data representation, see the “*Implementation Details*” chapter of the "*ST20 Embedded Toolset Reference Manual*".

Both functions require an STLite/OS20 priority level to be specified for the task and a name to be associated with the task for use by the debugger. The priority levels are defined in the header file `task.h` by the macros `OS20_PRIORITY_LEVEL`, `MAX_USER_PRIORITY` and `MIN_USER_PRIORITY`, see section 14.18.

For tasks running on an ST20-C2, both functions also enable the task to be elevated to a high priority process. In this case the STLite/OS20 task priority will not be used. High priority processes have restrictions associated with them as described in section 14.2.

14.5.1 Creating a task for an RCU

Two functions are provided for creating a task in a relocatable code unit: `task_create_sl` and `task_init_sl`. The chapter “*Building and running relocatable code*” in the "*ST20 Embedded Toolset Reference Manual*" provides details of using STLite/OS20 with relocatable code units.

14.6 Synchronizing tasks

Tasks synchronize their actions with each other using semaphores, as described in Chapter 15.

14.7 Communicating between tasks

Tasks communicate with each other by using message queues, as described in Chapter 16.

14.8 Timed delays

The following two functions cause a task to wait for a certain length of time as measured in ticks of the timer.

```
void task_delay(clock_t delay);
void task_delay_until(clock_t delay);
```

Both functions wait for a period of time and then return. `task_delay_until` waits until the given absolute reading of the timer is reached. If the requested time is before the present time, then the task does not wait.

`task_delay` waits until the given time has elapsed, that is, it delays execution for the specified number of timer ticks. If the time given is negative, no delay takes place.

`task_delay` or `task_delay_until` may be used for data logging or causing an event at a specific time. A high priority task can wait until a certain time; when it wakes it will preempt any lower priority task that is running and perform the time-critical function.

When initiating regular events, such as for data logging, it may be important not to accumulate errors in the time between ticks. This is done by repeatedly adding to a time variable rather than rereading the start time for the delay. For example, to initiate a regular event every `delay` ticks:

```
#include <ostime.h>

clock_t time;
time = time_now();
for (;;)
{
  time = time_plus (time, delay);
  task_delay_until(time);
  initiate_regular_event ();
}
```

14.9 Rescheduling

Sometimes, a task needs to voluntarily give up control of the CPU so that another task at the same priority can execute, that is, terminate the current timeslice. This may be achieved with the function:

```
void task_reschedule (void);
```

This provides a clean way of suspending execution of a task in favor of the next task on the scheduling list, but without losing priority. The task which executes `task_reschedule` is added to the back of the scheduling list and the task at the front of the scheduling list is promoted to be the new *current* task.

A task may be inadvertently rescheduled when the `task_priority_set()` function is used, see section 14.3.

14.10 Suspending tasks

Normally a task will only deschedule when it is waiting for an event, such as for a semaphore to be signalled. This requires that the task itself call a function indicating that it is willing to deschedule at that point (for example, by calling `semaphore_wait`). However, sometimes it is useful to be able to control a task, causing it to forcibly deschedule, without it explicitly indicating that it is willing to be descheduled. This can be done by *suspending* the task.

When a task is suspended, it will stop executing immediately. When the task should start executing again, another task must *resume* it. When it is resumed the task will be unaware it has been suspended, other than the time delay.

Task suspension is in addition to any other reason that a task is descheduled. Thus a task which is waiting on a semaphore, and which is then suspended, will not start executing again until both the task is resumed, and the semaphore is signalled, although these can occur in any order.

A task is suspended using the call:

```
int task_suspend(task_t* Task)
```

where `Task` is the task to be suspended. A task may suspend itself by specifying `Task` as `NULL`. The result is `0` if the task was successfully suspended, `-1` if it failed. This call will fail if the task has terminated. A task may be suspended multiple times by executing several calls to `task_suspend`. It will not start executing again until an equal number of `task_resume` calls have been made.

A task is resumed using the call:

```
int task_resume(task_t* Task)
```

where `Task` is the task to be resumed. The result is `0` if the task was successfully resumed, `-1` if it failed. The call will fail if the task has terminated, or is not suspended.

It is also possible to specify that when a task is created, it should be immediately suspended, before it starts executing. This is done by specifying the flag `task_flags_suspended` when calling `task_create` or `task_init`. This can be useful to ensure that initialization is carried out before the task starts running. The task is resumed in the usual way, by calling `task_resume`, and it will start executing from its entry point.

14.11 Killing a task

Normally a task runs to completion and then exits. It may also choose to exit early by calling `task_exit()`. However, it is also possible to force a task to exit early, using the function:

```
int task_kill(task_t* task, int status,
   task_kill_flags_t flags);
```

This will stop the task immediately, cause it to run the exit handler (if there is one), and exit.

Sometimes it may be desirable for a task to prevent itself being killed temporarily, for example, while it owns a mutual exclusion semaphore. To do this the task can make itself immortal by calling:

```
void task_immortal(void);
```

and once it is willing to be killed again calling:

```
void task_mortal(void);
```

While the task is immortal, it cannot be killed. However, if an attempt was made to kill the task whilst it was immortal, it will die immediately it makes itself mortal again by calling `task_mortal`.

Calls to `task_immortal` and `task_mortal` nest correctly, so the same number of calls need to be made to both functions before the task becomes mortal again.

14.12 Getting the current task's id

Several functions are provided for obtaining details of a specified task. The following function returns a pointer to the task structure of the current task:

```
task_t* task_id (void)
```

While `task_id` is very efficient when called from a task, it will take a long time to execute when called from a high priority process, and cannot be called from an interrupt handler. To avoid these problems an alternative function is available:

```
task_context_t task_context(task_t** task, int* level)
```

This will return whether it was called from a task, interrupt, or high priority process. In addition if `task` is not `NULL`, and `task_context` is called from a task or high priority process, it will assign the current task ID to the `task_t` pointed to by `task`. Similarly if `level` is not `NULL`, and `task_context` is called from an interrupt handler, then it will assign the current interrupt level to the `int` pointed to by `level`. The advantage in not requiring the current `task_t` or interrupt level is that this function may operate considerably faster when this information does not have to be found.

Both of these function may be used in conjunction with `task_wait`, see section 14.16.

The function:

```
const char*task_name(task_t *task);
```

returns the name of the specified task, or if `task` is NULL, the current task. (The task's name is set when the task is created).

14.13 Stack usage

A common problem when developing applications is not allocating enough stack for a task, or the need to tune stack allocation to minimize memory wastage. STLite/OS20 provides a couple of techniques which can be used to address this.

The first technique is to enable stack checking in the compiler see section 3.3.8. This adds an additional function call at the start of each of the user's functions, just before any additional stack is allocated. The called stack check function can then determine whether there is sufficient space available for the function which is about to execute.

As STLite/OS20 is multi-threaded, a special version of the stack check function needs to be used, which can determine the current task, and details about the task's stack. When using `-runtime os20` to link the application, the stack check function is linked in automatically. Otherwise it is necessary to link with the configuration file `os20scc1.cfg` (for a C1 target) or `os20scc2.cfg` (for a C2 target) to ensure the correct function is linked in.

Whilst stack checking has the advantage that a stack overflow is reported immediately it occurs, it has a number of problems:

- there is a run time cost incurred every function call to perform the check;
- it cannot report on functions which are not recompiled with stack checking enabled.

An alternative technique is to determine experimentally, how much stack a task uses by giving the task a large stack initially, running the code, and then seeing how much stack has been used. To support this STLite/OS20 normally fills a task's stack with a known value. As the task runs it will write its own data into the stack, altering this value, and later the stack can be inspected to determine the highest address which has not been altered.

To support this STLite/OS20 provides the function:

```
int task_status(task_t* Task, task_status_t *Status,
  task_status_flags_t Flags);
```

This function can be used to determine information about the task's stack, in particular the base and size specified when the task was created, and the amount of stack which has been used.

Stack filling is enabled by default, however, in some cases the user may want to control it, so two functions are provided:

```
int task_stack_fill(task_stack_fill_t* fill);
```

returns details about the current stack fill settings, and:

```
int task_stack_fill_set(task_stack_fill_t* fill);
```

allows them to be altered. Stack filling can be enabled or disabled, or the fill value changed. By default it is enabled, and the fill value set to `0x12345678`.

By placing a call to `task_stack_fill_set` in a start-up function, before the STLite/OS20 kernel is initialized, it is possible to control the filling of the root task's stack.

To determine how much stack has been used `task_status` can be called, with the `Flags` parameter set to `task_status_flags_stack_used`. For this to work correctly, task stack filling must have been enabled when the task was created, and the fill value must have the same value as the one which was in effect when the task was created.

14.14 Task data

STLite/OS20 provides one word of '*task-data*' *per* task. This can be used by the application to store data which is specific to the task, but which needs to be accessed uniformly from multiple tasks.

This is typically used to store data which is required by a library, when the library can be used from multiple tasks but the data is specific to the task. For example, a library which manages an I/O channel may be called by multiple tasks, each of which has its own I/O buffers. To avoid having to pass an I/O descriptor into every call it could be stored in task-data.

Although only one word of storage is provided, this is usually treated as a pointer, which points to a user defined data structure which can be as large as required.

Two functions provide access to the task-data pointer:

```
void* task_data_set (task_t* Task, void* NewData);
```

`task_data_set()` sets the task-data pointer of the task specified by `Task`.

```
void* task_data(task_t* Task);
```

`task_data()` retrieves the task-data pointer of the task specified by `Task`.

If `Task` is `NULL` both functions use the currently active task.

When a task is first created (including the root task), its task-data pointer is set to NULL (0). For example:

```
typedef struct {
  char   buffer[BUFFER_SIZE];
  char*  buffer_next;
  char*  buffer_end;
} ptd_t;

char buffer_read(void)
{
  ptd_t *ptd;

  ptd = task_data(NULL);
  if (ptd->buffer_next == ptd->buffer_end) {
    ... fill buffer ...
  }
  return *(ptd->buffer_next++);
}

int main()
{
  ptd_t *ptd;
  task_t *task;

  ... create a task ...

  ptd = memory_allocate(system_partition, sizeof(ptd_t));
  ptd->buffer_next = ptd->buffer_end = ptd->buffer;
  task_data_set(task, ptd);

}
```

14.15 Task termination

A task terminates when it returns from the task's entry point function.

A task may also terminate by using the following function:

```
void task_exit(int param);
```

In both cases an exit status can be specified. When the task returns from its entry point function, the exit status is the value that the function returns. If `task_exit` is called then the exit status is specified as the parameter. This value is then made available to the '*onexit*' handler if one has been installed (see below).

Just before the task terminates (either by returning from its entry point function, or calling `task_exit`), it will call an '*onexit*' handler. This function allows any application specific tidying up to be performed before the task terminates. The *onexit* handler is installed by calling:

```
task_onexit_fn_t task_onexit_set(task_onexit_fn_t fn);
```

The *onexit* handler function must have a prototype of:

```
void onexit_handler(task_t *task, int param)
```

When the handler function is called, `task` specifies the task which has exited, and `param` is the task's exit status.

The function `task_onexit_set_sl` is provided to set the task *onexit* handler and specify a static link.

The following code example shows how a task's exit code can be stored in its task-data (see section 14.14), and retrieved later by another task which is notified of the termination through `task_wait`.

```
void onexit_handler(task_t* task, int param)
{
    task_data_set(NULL, (void*)param);
}

int main()
{
  task_t *Tasks[NO_USER_TASKS];

  /* Set up the onexit handler */
  task_onexit_set(onexit_handler);

  ... create the tasks ...

  /* Wait for the tasks to finish */
  for (i=0; i<NO_USER_TASKS; i++) {
    int t;
    t = task_wait(Tasks, NO_USER_TASKS, TIMEOUT_INFINITY);
    printf("Task %d : exit code %d\n", t, (int)task_data(Tasks[t]));
    Tasks[t] = NULL;
  }
}
```

14.16 Waiting for termination

It is only safe to free or otherwise reuse a task's stack, once it has terminated.

The following function waits until one of a list of tasks terminates or the specified timeout period is reached:

```
int task_wait(task_t **tasklist, int ntasks,
   const clock_t *timout);
```

Timeouts for tasks are implemented using hardware and so do not increase the application's code size. Any task can wait for any other asynchronous task to complete. A parent task should, for example, wait for any children to terminate. In this case `task_wait` can be used inside a loop.

After `task_wait` has indicated that a particular task has completed, any of the task's data including any memory dynamically loaded or allocated from the heap and used for the task's stack, can be freed. The task's state: its control block `task_t` and descriptor `tdesc_t` may also be freed. (`task_delete` can be used to free `task_t` and `tdesc_t`, see section 14.17).

The `timeout` period for `task_wait` may be expressed as a number of ticks or it may take one of two values: `TIMEOUT_IMMEDIATE` indicates that the function should return immediately, even if no tasks have terminated, and `TIMEOUT_INFINITY` indicates that the function should ignore the timeout period, and only return when a task terminates. The header file `ostime.h` must be included when using this function.

14.17 Deleting a task

A task can be deleted by using the `task_delete` function:

```
#include <task.h>
int task_delete(task_t* task);
```

This will remove the task from the list of known tasks and allow its stack and data structures to be reused.

If the task was created using `task_create` then `task_delete` calls `memory_deallocate` in order to free the task's state (both static `task_t` and dynamic `tdesc_t`) and the task's stack.

A task must have terminated before it can be deleted, if it has not `task_delete` will fail.

14.18 Task header file: task.h

All the definitions related to tasks are in the single header file, `task.h`, see Table 14.1, Table 14.2 and Table 14.3.

Function	Description
`task_context`	Return the current execution context.
`task_create`	Create an STLite/OS20 task.
`task_create_sl`	Create an STLite/OS20 task specifying a static link.
`task_data`	Retrieve a task’s data pointer.
`task_data_set`	Sets a task’s data pointer.
`task_delay`	Delay the calling task for a period of time.
`task_delay_until`	Delay the calling task until a specified time.
`task_delete`	Delete a task.
`task_exit`	Exits the current task.
`task_id`	Find current task’s id.
`task_immortal`	Make the current task immortal.
`task_init`	Initialize an STLite/OS20 task.
`task_init_sl`	Initialize an STLite/OS20 task specifying a static link.
`task_kill`	Kill a task.
`task_lock`	Prevent task rescheduling.
`task_mortal`	Make the current task mortal.
`task_name`	Returns the task’s name.
`task_onexit_set`	Setup a function to be called when a task exits.
`task_onexit_set_sl`	Setup a function to be called when a task exits and specify a static link.
`task_priority`	Retrieve a task’s priority.
`task_priority_set`	Set a task’s priority.
`task_reschedule`	Reschedule the current task.
`task_resume`	Resume a suspended task.
`task_suspend`	Suspend a task.
`task_stack_fill`	Return the task fill configuration.
`task_stack_fill_set`	Set the task stack fill configuration.
`task_status`	Return status information about the task.
`task_unlock`	Allow task rescheduling.
`task_wait`	Waits until one of a list of tasks completes.

Table 14.1 Functions defined in `task.h`

.

Types	Description
task_context_t	Result of task_context.
task_flags_t	Additional flags for task_create and task_init.
task_kill_flags_t	Additional flags for task_kill.
task_onexit_fn_t	Function to be called on task exit.
task_state_t	State of a task (for example: active, deleted).
task_stack_fill_state_t	Whether stack filling is enabled or disabled.
task_stack_fill_t	Stack filling state (specifies enables and value).
task_status_flags_t	Additional flags for task_status.
task_status_t	Result of task_status.
task_t	A task’s static state.
tdesc_t	A task’s dynamic state.

Table 14.2 Types defined in task.h

Macro	Description
OS20_PRIORITY_LEVELS	Number of STLite/OS20 task priorities. Default is 16.
MAX_USER_PRIORITY	Highest user task priority. Default is 15.
MIN_USER_PRIORITY	Lowest user task priority. Default is 0.

Table 14.3 Macros defined in task.h

15 Semaphores

Semaphores provide a simple and efficient way to synchronize multiple tasks. Semaphores can be used to ensure mutual exclusion, control access to a shared resource, and synchronize tasks.

15.1 Overview

A semaphore structure `semaphore_t` contains two pieces of data:

- A count of the number of times the semaphore can be taken.
- A queue of tasks waiting to take the semaphore.

Semaphores are created using one of the following functions:

```
semaphore_t* semaphore_create_fifo (int value);
void semaphore_init_fifo(semaphore_t *sem, int value);
semaphore_t* semaphore_create_priority (int value);
void semaphore_init_priority (semaphore_t *sem, int value);
```

or if a timeout capability is required while waiting for a semaphore, use the timeout versions of the above functions:

```
semaphore_t* semaphore_create_fifo_timeout (int value);
void semaphore_init_fifo_timeout(
                          semaphore_t *sem, int value);
semaphore_t* semaphore_create_priority_timeout (int value);
void semaphore_init_priority_timeout (
                           semaphore_t *sem, int value);
```

The `create_` versions of the functions will allocate memory for the semaphore automatically, while the `init_` versions enable the user to specify a pointer to the semaphore, using the data structure `semaphore_t`.

The semaphores which STLite/OS20 provides differ in the way in which tasks are queued. Normally tasks are queued in the order which they call `semaphore_wait`, in which case this is termed a FIFO semaphore. Semaphores of this type are created using `semaphore_create_fifo` or `semaphore_init_fifo` or by using one of the timeout versions of these functions.

However, sometimes it is useful to allow higher priority tasks to jump the queue, so that they will be blocked for a minimum amount of time. In this case a second type of semaphore can be used, a priority based semaphore. For this type of semaphore, tasks will be queued based on their priority first, and the order which they call `semaphore_wait` second. Semaphores of this type are created using `semaphore_create_priority` or `semaphore_init_priority` or one of the timeout versions of these functions.

Semaphores may be acquired by the function:

```
void semaphore_wait (semaphore_t* Sem);
```

For semaphores created via one of the timeout functions, then the following function may also be used:

```
int semaphore_wait_timeout(
    semaphore_t* Sem
    const clock_t *timeout);
```

When a task wants to acquire a semaphore, it calls `semaphore_wait`. At this point if the semaphore count is greater than 0, then the count will be decremented, and the task continues. If however, the count is already 0, then the task will add itself to the queue of tasks waiting for the semaphore and deschedule itself. Eventually another task should release the semaphore, and the first waiting task will be able to continue. In this way, when the task returns from the function it will have acquired the semaphore.

If you want to make certain that the task does not wait indefinitely for a particular semaphore then the timeout versions of the semaphore functions may be used. **Note** that these functions cannot use the hardware support for semaphores, and so are larger and slower than the non-timeout versions.

`semaphore_wait_timeout` enables a timeout to be specified. If this time is reached before the semaphore is acquired then the function will return and the task continues without acquiring the semaphore. Two special values may be specified for the timeout period:

- `TIMEOUT_IMMEDIATE` causes the semaphore to be polled and the function to return immediately. The semaphore may or may not be acquired and the task continues.
- `TIMEOUT_INFINITY` causes the function to behave the same as `semaphore_wait`, that is, the task will wait indefinitely for the semaphore to become available.

When a task wants to release the semaphore, it calls `semaphore_signal`:

```
void semaphore_signal (semaphore_t* Sem);
```

This will look at the queue of waiting tasks, and if the queue is not empty, remove the first task from the queue, and start it running. If there are no tasks waiting, then the semaphore count will be incremented, indicating that the semaphore is available.

If a semaphore is deleted using `semaphore_delete` then how the memory is released will depend on whether the semaphore was created by the `create` or `init` version of the function. See the functional description of `semaphore_delete` in “*Part 4 - STLite/OS20 functions*” in the "*ST20 Embedded Toolset Reference Manual*".

An important use of semaphores is for synchronization between interrupt handlers and tasks. This is possible because while an interrupt handler cannot call `semaphore_wait`, it can call `semaphore_signal`, and so cause a waiting task to start running.

FIFO semaphores can also be used to synchronize the activity of low priority tasks with high priority tasks.

15.2 Use of Semaphores

Semaphores can be defined to allow a given number of tasks simultaneous access to a shared resource. The maximum number of tasks allowed is determined when the semaphore is initialized. When that number of tasks have acquired the resource, the next task to request access to it will wait until one of those holding the semaphore relinquishes it.

Semaphores can protect a resource only if all tasks that wish to use the resource also use the same semaphore. It cannot protect a resource from a task that does not use the semaphore and accesses the resource directly.

Typically, semaphores are set up to allow at most one task access to the resource at any given time. This is known as using the semaphore in *binary mode*, where the count either has the value zero or one. This is useful for mutual exclusion or synchronization of access to shared data. Areas of code protected using semaphores are sometimes called *critical regions*.

When used for mutual exclusion the semaphore is initialized to one, indicating that no task is currently in the critical region, and that at most one can be. The critical region is surrounded with calls to `semaphore_wait` at the start and `semaphore_signal` at the end. Thus the first task which tries to enter the critical region will successfully take the semaphore, and any others will be forced to wait. When the task currently in the critical region leaves, it releases the semaphore, and allows the first of the waiting tasks into the critical region.

Semaphores are also used for synchronization. Usually this is between a task and an interrupt handler, with the task waiting for the interrupt handler. When used in this way the semaphore is initialized to zero. The task then performs a `semaphore_wait` on the semaphore, and will deschedule. Later the interrupt handler will perform a `semaphore_signal`, which will reschedule the task. This process can then be repeated, with the semaphore count never changing from zero.

All the STLite/OS20 semaphores can also be used in a *counting* mode, where the count can be any positive number. The typical application for this is controlling access to a shared resource, where there are multiple resources available. Such a semaphore allows *N* tasks simultaneous access to a resource and is initialized with the value *N*. Each task performs a `semaphore_wait` when it wants a device. If a device is available the call will return immediately having decremented the counter. If no devices are available then the task will be added to the queue. When a task has finished using a device it calls `semaphore_signal` to release it.

15.3 Semaphore header file: semaphor.h

All the definitions related to semaphores are in the single header file, `semaphor.h`, see Table 15.1 and Table 15.2.

Function	Description
`semaphore_create_fifo`	Create a FIFO queued semaphore
`semaphore_create_fifo_timeout`	Create a FIFO queued semaphore with timeout
`semaphore_create_priority`	Create a priority queued semaphore
`semaphore_create_priority_timeout`	Create a priority queued semaphore with timeout
`semaphore_delete`	Delete a semaphore
`semaphore_init_fifo`	Initialize a FIFO queued semaphore
`semaphore_init_fifo_timeout`	Initialize a FIFO queued semaphore with timeout
`semaphore_init_priority`	Initialize a priority queued semaphore
`semaphore_init_priority_timeout`	Initialize a priority queued semaphore with timeout
`semaphore_signal`	Signal a signal
`semaphore_wait`	Wait for a signal
`semaphore_wait_timeout`	Wait for a semaphore or a timeout

Table 15.1 Functions defined in `semaphor.h`

Types	Description
`semaphore_t`	A semaphore

Table 15.2 Types defined in `semaphor.h`

16 Message handling

A message queue provides a buffered communication method for tasks. Message queues also provide a way to communicate without copying the data, which can save time. Message queues are, however, subject to the following restriction:

- Message queues may only be used from interrupt handlers if the timeout versions of the message handling functions are used and a timeout period of `TIMEOUT_IMMEDIATE` is used, see section 16.3. This prevents the interrupt handler from blocking on a message claim.

16.1 Message queues

An STLite/OS20 message queue implements two queues of messages, one for message buffers which are currently not being used (known as the '*free*' queue), and the other holds messages which have been sent but not yet received (known as the '*send* ' queue). Message buffers rotate between these queues, as a result of the user calling the various message functions.

The movement of messages between the two queues is illustrated in Figure 16.1.

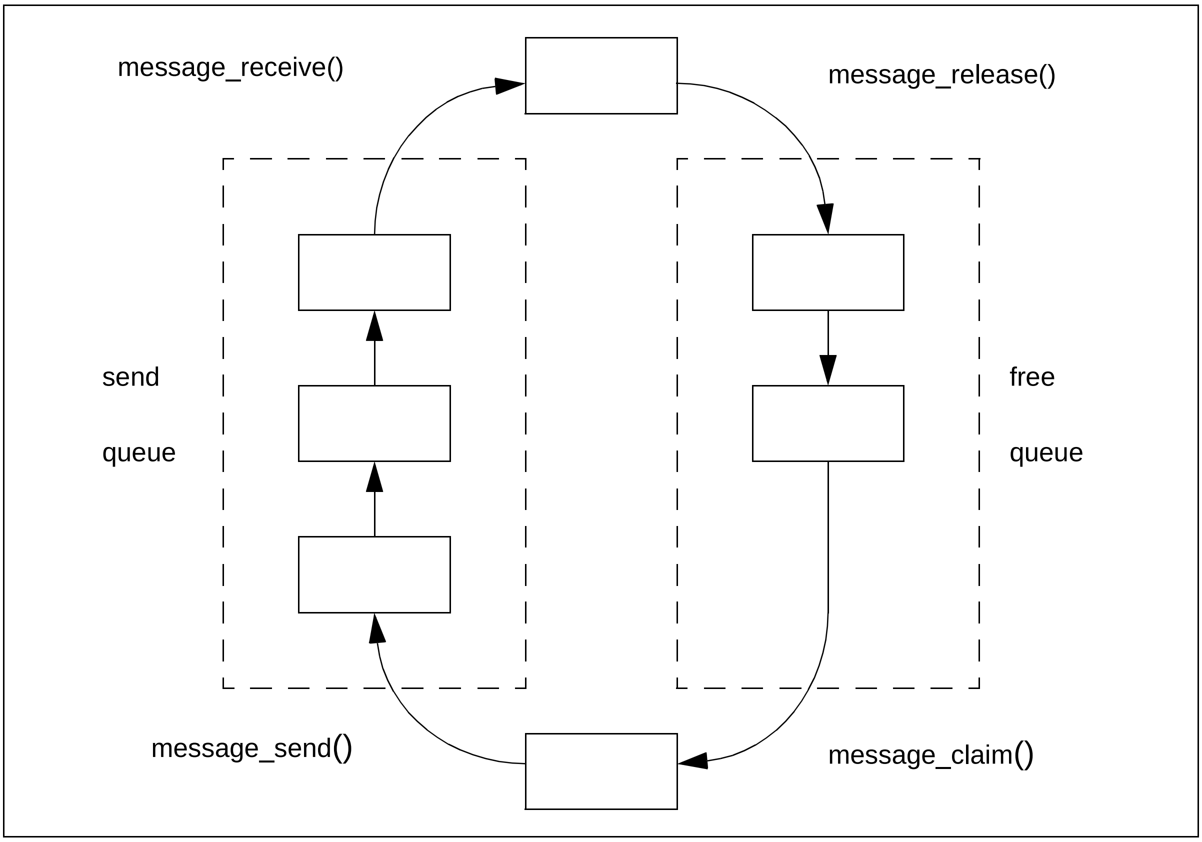


Figure 16.1 Message queues

16.2 Creating message queues

Message queues are created using one of the following functions:

```
#include <message.h>

message_queue_t* message_create_queue(
  size_t MaxMessageSize,
  unsigned int MaxMessages);
```

```
#include <message.h>

void message_init_queue (
  message_queue_t* MessageQueue,
  void* memory,
  size_t MaxMessageSize,
  unsigned int MaxMessages);
```

or by using timeout versions of the above functions:

```
#include <message.h>

message_queue_t* message_create_queue_timeout(
  size_t MaxMessageSize,
  unsigned int MaxMessages);

#include <message.h>

void message_init_queue_timeout(
  message_queue_t* MessageQueue,
  void* memory,
  size_t MaxMessageSize,
  unsigned int MaxMessages);
```

These functions create a message queue for a fixed number of fixed sized messages, each message being preceded by a header, see Figure 16.2. The user must specify the maximum size for a message element and the total number of elements required.

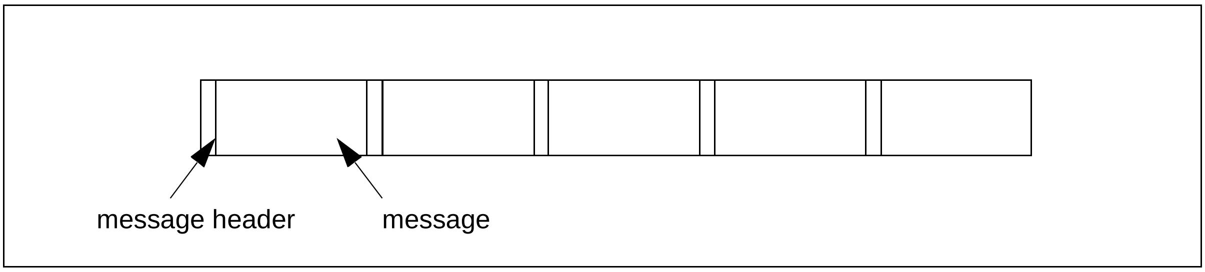


Figure 16.2 STLite/OS20 message elements

`message_create_queue` and `message_create_queue_timeout` allocates the memory for the queue automatically from the system partition.

`message_init_queue` and `message_init_queue_timeout` requires the user to allocate the memory for the message queue. This needs to be large enough for storing all the messages (rounded up to the nearest word size) plus a header, for each message.

The total amount of memory needed (in bytes) can be calculated using the macro:

```
MESSAGE_MEMSIZE_QUEUE(maxMessageSize, maxMessages)
```

where `maxMessageSize` is the size of the message, and `maxMessages` is the number of messages.

As long as both of the parameters can be determined at compile time, this macro can be completely evaluated at compile time, and so can be used as the dimension of an array, for example:

```
typedef struct {
  int tag;
  char msg[10];
} msg_t;
#define NUM_MSG 10
char msg_buffer[MESSAGE_MEMSIZE_QUEUE(sizeof(msg_t), NUM_MSG);
```

Alternatively this can be done by calling the function `memory_allocate`. This function will return a pointer to the allocated memory, which should be passed to `message_init_queue` or `message_init_queue_timeout` as the parameter `MessageQueue`.

Note that these functions cannot use the hardware support for semaphores, and so are larger and slower than the non-timeout versions.

Example

```
#include <message.h>
#include <partitio.h>

#define MSG_SIZE 512
#define MAX_MSGS 10

#define QUEUE_SIZE MESSAGE_MEMSIZE_QUEUE(MSG_SIZE,MAX_MSGS)

#define EXIT_SUCCESS 0
#define EXIT_FAILURE -1

int myqueue_create(void)
{
  void *msg_queue;
  message_queue_t *msg_queue_struct;


  /* allocate memory for message queue itself */

  msg_queue = memory_allocate(system_partition,QUEUE_SIZE);
  if (msg_queue == 0)
    {
      return(EXIT_FAILURE);
    }

/* allocate memory for message struct which holds details of queue */

  msg_queue_struct = memory_allocate(system_partition,sizeof(
                                          message_queue_t));
  if (msg_queue_struct == 0)
    {
      memory_deallocate(system_partition,msg_queue);
      return(EXIT_FAILURE);
    }

  message_init_queue(msg_queue_struct,msg_queue,MSG_SIZE,MAX_MSGS);
  return(EXIT_SUCCESS);
}
```

16.3 Using message queues

Initially all the messages are on the free queue. The user allocates free message buffers by calling either of the following functions, which can then be filled in with the required data:

```
void* message_claim(              void* message_claim_timeout(
    message_queue_t* queue);          message_queue_t* queue
                                      const clock_t* time);
```

Both functions claim the next available message in the message queue. `message_claim_timeout` enables a timeout to be specified but can only be used if the message queue was created with a timeout capability. If the timeout is reached before a message buffer is acquired then the function will return NULL. Two special values may be specified for the timeout period:

- `TIMEOUT_IMMEDIATE` causes the message queue to be polled and the function to return immediately. A message buffer may or may not be acquired and the task will continue.
- `TIMEOUT_INFINITY` causes the function to behave the same as `message_claim`, that is, the task will wait indefinitely for a message buffer to become available.

When the message is ready it is sent by calling `message_send()`, at which point it is added to the send queue.

Messages are removed from the send queue by a task calling either of the functions:

```
void* message_receive(            void* message_receive_timeout(
    message_queue_t* queue);          message_queue_t* queue
                                      const clock_t* time);
```

Both functions return the next available message. `message_receive_timeout` provides a timeout facility which behaves in a similar manner to `message_claim_timeout` in that it returns NULL if message does not become available. If `TIMEOUT_IMMEDIATE` is specified the task will continue whether or not a message is received and if `TIMEOUT_INFINITY` is specified the function will behave as `message_receive` and wait indefinitely.

Finally when the receiving task has finished with the message buffer it should free it by calling `message_release()`, which will add it to the free queue, where it is again available for allocation.

If the size of the message is variable, the user should specify that the message is `sizeof(void*)`, and then use pointers to the messages as the arguments to the message functions. The user is then responsible for allocating and freeing the real messages using whatever techniques are appropriate.

Message queues may be deleted by calling `message_delete_queue()`. If the message queue was created using `message_create_queue` or `message_create_queue_timeout` then this will also free the memory allocated for the message queue. If it was created using `message_init_queue` or `message_init_queue_timeout` then the user is responsible for freeing any memory which was allocated for the queue.

16.4 Message header file: message.h

All the definitions related to messages are in the single header file, `message.h`, see Table 16.1 and Table 16.2.

Function	Description
`message_claim`	Claim a message buffer
`message_claim_timeout`	Claim a message buffer with timeout
`message_create_queue`	Create a fixed size message queue
`message_create_queue_timeout`	Create a fixed size message queue with timeout
`message_delete_queue`	Delete a message queue
`message_init_queue`	Initialize a fixed size message queue
`message_init_queue_timeout`	Initialize a fixed size message queue with timeout
`message_receive`	Receive the next available message from a queue
`message_receive_timeout`	Receive the next available message from a queue or timeout
`message_release`	Release a message buffer
`message_send`	Send a message to a queue

Table 16.1 Functions defined in `message.h`

Types	Description
`message_hdr_t`	A message buffer header
`message_queue_t`	A message queue

Table 16.2 Types defined in `message.h`

17 Real-time clocks

Time is very important for real-time systems. STLite/OS20 provides some basic functions for manipulating quantities of time:

The ST20 traditionally regards time as circular. That is, the counters which represent time can wrap round, with half the time period being in the future, and half of it in the past. This behavior means that clock values should only be manipulated using time functions. STLite/OS20 provides functions to:

- Add and subtract quantities of time.
- Determine if one time is after another.
- Return the current time.

17.1 ST20-C1 clock peripheral

The ST20-C1 microprocessor does not have its own clock so a clock peripheral is required when using STLite/OS20.

A number of functions are required to support the clock functions provided by STLite/OS20. STLite/OS20 provides the sources for a number of library functions to support a clock peripheral running on an ST20-MC2 device. These functions may be copied and modified to support other clock peripherals, see Chapter 21.

Note the STLite/OS20 kernel and interrupt controller must be initialized (as described in Chapter 21) before the clock peripheral is initialized.

17.2 The ST20 timers on the ST20-C2

An ST20-C2 processor has two on-chip real-time 32-bit clocks, called timers, one with low resolution and one with high resolution. The following details are relevant for some ST20-C2 devices. You should check the figures given in the device datasheet as the timing values vary with different processor revisions.

The low resolution clock can be used for timing periods up to approximately 38 hours, with a resolution of 64 µsec. The low resolution clock is accessed by low priority tasks. The high resolution clock can be used for timing periods up to approximately half an hour with a resolution of 1 µsec. The high resolution clock is accessed by high priority tasks. Longer periods can be timed with either timer by explicitly incrementing a counter.

The clocks start at an undefined value and wrap round to 0 on the next tick after hexadecimal FFFFFFFF, or, if treated as signed, to the most negative integer on the next tick after the most positive integer. The tick rate of the clocks is derived from the processor input **ClockIn**, and the speed and accuracy depends on the speed and accuracy of the input clock.

For ST20 variants with power-down capability, the clocks pause when the ST20 is in power-down mode.

	Low priority	**High priority**
Interval between ticks	64 µsec	1 µsec
Ticks per second	15625	1000000
Approximate full timer cycle	76.35 hours	1.193 hours

Table 17.1 Summary of clock intervals for parts operating at 40Mhz

17.3 Reading the current time

The value of a timer (or clock) is read using `time_now` which returns the value of the timer for the current priority.

```
#include <ostime.h>
clock_t time_now (void);
```

The time at which counting starts will be no later than the call to `kernel_start`.

17.4 Time arithmetic

Arithmetic on timer values should always be performed using special modulo operators. These routines perform no overflow checking and so allow for timer values ‘wrapping round’ to the most negative integer on the next tick after the most positive integer.

```
clock_t time_plus(const clock_t time1, const clock_t time2);
clock_t time_minus(const clock_t time1, const clock_t time2);
int time_after(const clock_t time1, const clock_t time2);
```

`time_plus` adds two timer values together and returns the sum allowing for any wrap-around. For example, if a number of ticks is added to the current time using `time_plus` then the result is the time after that many ticks.

`time_after` subtracts the second value from the first and returns the difference allowing for any wrap-around. For example, if one time is subtracted from another using `time_after` then the result will be the number of ticks between the two times. If the result is positive then the first time is after the second. If the result is negative then the first time is before the second.

`time_after` determines whether the first time is after the second time. One time is considered to be after another if the one is not more than half a full timer cycle later than the other. Half a full cycle is 2^{31} ticks. The function returns the integer value `1` if the first time is after the second, and otherwise it returns zero.

Some of these concepts are shown in Figure 17.1.

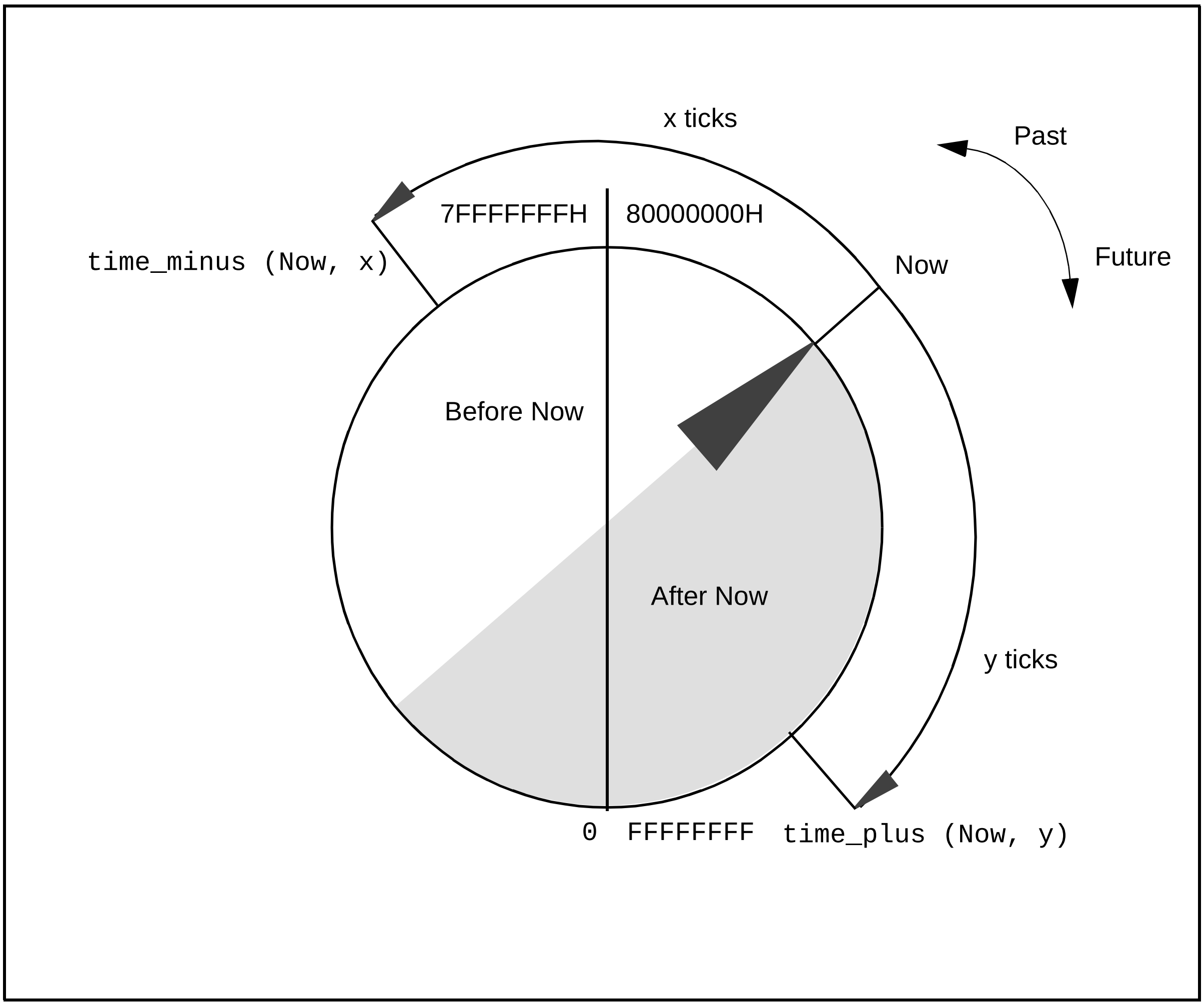


Figure 17.1 Time arithmetic

Time arithmetic is modulo 2^{32}. In applications running for a long time, some care must be taken to ensure that times are close enough together for arithmetic to be meaningful. For example, subtracting two times which are more than 2^{31} ticks apart will produce a result that must be interpreted with care. Very long intervals can be tracked by counting a number of cycles of the clock.

17.5 Time header file: ostime.h

All the definitions related to time are in the single header file, `ostime.h`, see Table 17.2.

Function	Description
`time_after`	Return whether one time is after another
`time_minus`	Subtract two clock values
`time_now`	Return the current time
`time_plus`	Add two clock values

Table 17.2 Functions defined in `ostime.h`

Table 17.3 lists the types defined by `ostime.h`.

Types	Description
`clock_t`	Number of processor clock ticks

Table 17.3 Types defined by `ostime.h`

18 Interrupts

Interrupts provide a way for external events to control the CPU. Normally, as soon as an interrupt is asserted, the CPU will stop executing the current task, and start executing the interrupt handler for that interrupt. In this way the program can be made aware of external changes as soon as they occur. This switch is performed completely in hardware, and so can be extremely rapid. Similarly when the interrupt handler has completed, the CPU will resume execution of the interrupted task, which will be unaware that it has been interrupted.

The interrupt handler which the CPU executes in response to the interrupt is called the first level interrupt handler. This piece of code is supplied as part of STLite/OS20, and simply sets up the environment so that a normal C function can be called. The STLite/OS20 API allows a different user function to be associated with each interrupt, and this will be called when the interrupt occurs. Each interrupt also has a parameter associated with it, which will be passed into the function when it is called. This could be used to allow the same code to be shared between different interrupt handlers.

18.1 Interrupt models

The interrupt hardware on different ST20 processors is similar, but there are a number of variations.

The basic hardware unit is called the *interrupt controller*. This receives the interrupt signals, and alerts the CPU when interrupts go active. Interrupts can be programmed to be active when high, or low, or on a rising, falling or both edges of the signal, this is called the *trigger mode* by STLite/OS20.

On some processors, interrupt sources are connected directly to the interrupt controller, similar to the example shown in Figure 18.1.

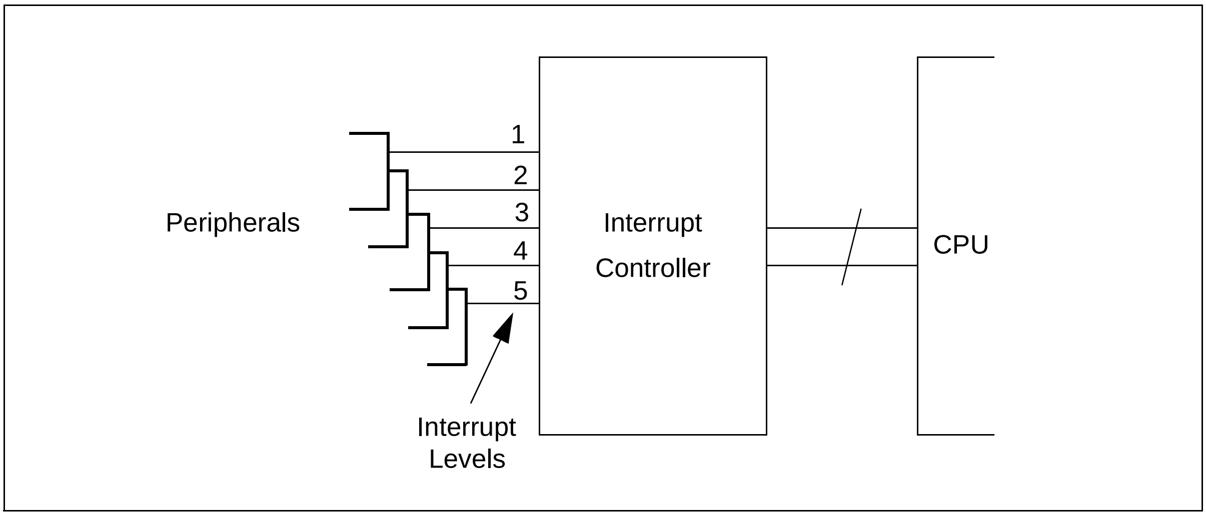


Figure 18.1 Example: peripherals directly attached to the interrupt controller

The relative priority of the interrupts is defined by the *interrupt level*, with numerically higher interrupts interrupting numerically lower priority interrupts. Thus, an interrupt level 3 will interrupt an interrupt level 2 which will interrupt an interrupt level 1. As the connection between the peripheral and the interrupt controller is fixed when the device is designed, so is the relative priority of the peripheral's interrupts.

Some ST20 processors have a second piece of interrupt hardware, called the interrupt level controller, see the example in Figure 18.2. This allows the relative priority of different interrupt sources to be changed. Each peripheral generates an *interrupt number*, which is fixed for the peripheral. This is fed into the interrupt level controller, which selects for each interrupt number which *interrupt level* should be generated. The interrupt level controller can be programmed to select the interrupt level which each interrupt number generates, thus allowing the relative priorities to be changed in software. As there are generally more interrupt numbers than interrupt levels, it is possible to multiplex several interrupt numbers onto a single interrupt level.

An important distinction is that interrupt levels are prioritized, numerically higher interrupt levels preempt lower ones, however there is no order between interrupt numbers.

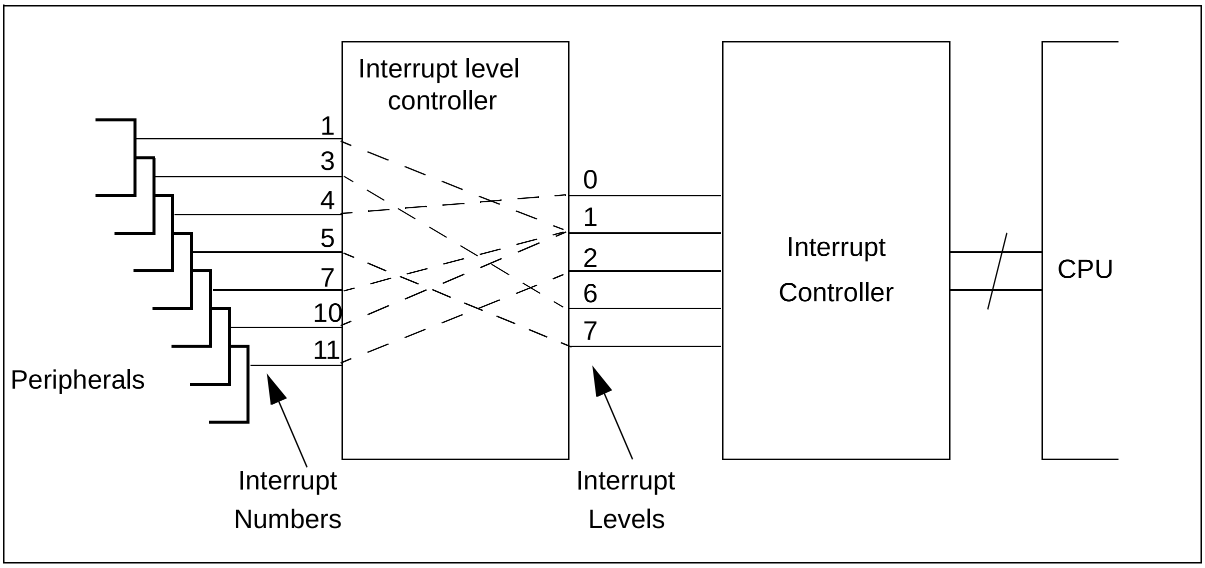


Figure 18.2 Example: peripherals mapped via an interrupt level controller

There are two types of interrupt controller for ST20 processors: IntC-1 and IntC-2. Both interrupt controllers provide the same services but the IntC-2 has a register layout that makes it capable of supporting more interrupt levels in the future, see Table 18.1.

There are three types of interrupt level controller for ST20 processors: ILC-1, ILC-2 and ILC-3.

ILC-1 type interrupt level controllers support up to 32 interrupt numbers and the trigger mode logic is part of the interrupt controller. All interrupt numbers attached to the same level share the same trigger mode.

ILC-2 type interrupt level controllers support up to 32 interrupt numbers but there is support for programmable trigger modes and an enable and disable facility for all interrupt numbers.

ILC-3 type interrupt level controllers currently supports up to 128 interrupt numbers, each of which can have a programmable trigger mode and enable status.

STLite/OS20 functions provide support for all ST20 interrupt models.

18.2 Selecting the correct interrupt handling system

STLite/OS20 contains two libraries to support different interrupt controller combinations:

Library	Description	Devices
`os20intc1.lib`	IntC-1	ST20GP6, ST20MC2, ST20TP3, ST20TP4, STV3500, STV0396, STi5500, STi5505, STi5508, STi5510, STi5512, STi5514, STi5516, STi5518, STi5519, STi5580, ST20-C1 simulator, ST20-C2 simulator
`os20intc2.lib`	IntC-2	ST20DC1, ST20DC2.

Table 18.1 Interrupt controller libraries

Additionally STLite/OS20 contains four libraries to support different interrupt level controller combinations:

Library	Description	Devices
`os20ilcnone.lib`	ILC-None	ST20-C1 simulator, ST20-C2 simulator.
`os20ilc1.lib`	ILC-1	ST20DC1, ST20DC2, ST20GP6, ST20MC2, ST20TP3, ST20TP4, STi5500, STi5505, STi5508, STi5510, STi5512, STi5580
`os20ilc2.lib`	ILC-2	STi5518, STi5519
`os20ilc2b.lib`	ILC-2B	STV0396
`os20ilc3.lib`	ILC-3	STi5514, STi5516, STV3500

Table 18.2 Interrupt level controller libraries

In order for STLite/OS20 to operate properly the correct libraries must be linked in. When using the `st20cc -runtime os20` option, the linker needs to select the appropriate IntC and ILC libraries. When using the `chip` command, the correct libraries will always be selected. If the `chip` command is not used then IntC-1 and ILC-1 libraries will be used to preserve backward compatibility.

In addition to providing support for the ILC-1, the ILC-1 library can support systems without an interrupt level controller and systems that have an ILC-2. In both these cases the support is not optimal. The ILC-None library uses much less RAM than its ILC-1 counterpart. The ILC-2 library supports the extra features the ILC-2 provides.

Note that the interrupt function definitions given in this chapter, list the interrupt level controllers they can be used with. If a function is used which is not applicable to the interrupt level controller on the device used then that function will not be provided and the application will fail at link time. This can cause link errors if the interrupt calls are used inappropriately. There are no warnings issued at compile time.

18.2.1 Compiling legacy code

The IntC-1 and IntC-2 libraries provide an identical set of function calls. There are no problems compiling code for either interrupt controller.

On ILC-2 and ILC-3 interrupt level controllers new function calls have been introduced to provide support for the newer features of these controllers. This may cause problems when reusing existing code. In particular be aware that calls to `interrupt_enable` and `interrupt_disable` should be replaced with calls to `interrupt_enable_number` and `interrupt_disable_number`. Code that does not do this will compile and link cleanly but interrupts will never be serviced because they are not enabled. Table 18.3 describes the migration path between ILCs.

<table>
<tr><th>ILC library</th><th>Legacy code</th><th>Recommended replacement</th></tr>
<tr><td rowspan="3">ILC-1</td><td><code>interrupt_enable (INTERRUPT_GLOBAL_ENABLE)</code></td><td><code>interrupt_enable_global()</code> †</td></tr>
<tr><td><code>interrupt_disable (INTERRUPT_GLOBAL_DISABLE)</code></td><td><code>interrupt_disable_global()</code> †</td></tr>
<tr><td><code>interrupt_pending_number</code></td><td><code>interrupt_test_number</code> †</td></tr>
<tr><td rowspan="3">ILC-2</td><td colspan="2">All changes recommended for ILC-1 plus:</td></tr>
<tr><td><code>interrupt_enable</code></td><td><code>interrupt_enable_number</code> ‡
(May require multiple calls.)</td></tr>
<tr><td><code>interrupt_disable</code></td><td><code>interrupt_disable_number</code> ‡
(May require multiple calls.)</td></tr>
<tr><td rowspan="2">ILC-3</td><td colspan="2">All changes recommended for ILC-2 plus:</td></tr>
<tr><td><code>interrupt_pending_number</code></td><td><code>interrupt_test_number</code> ‡</td></tr>
<tr><td colspan="3">† This change is optional and will make code easier to port in the future.</td></tr>
<tr><td colspan="3">‡ This change is mandatory on the ILC-3 and the default ILC-2 library. See also the Note below.</td></tr>
</table>

Table 18.3 Migration path for ILCs

Note: that at reset the ILC-2 hardware is configured to be backwards compatible with the ILC-1. The fastest way to bring old code up on these chips is to link in the ILC-1 library instead of the ILC-2 support. An STLite/OS20 configuration option is provided to support this when `st20cc -runtime os20` is used. See the chapter “*Advanced configuration of STLite/OS20*” in the "*ST20 Embedded Toolset Reference Manual*".

18.2.2 Linking legacy code

The only method of linking that is recommended is to use the command `st20cc -runtime os20` in conjunction with the application using the `chip` command. New programs should all follow this methodology.

Linking STLite/OS20 applications using `st20cc -Tos20.cfg` is not recommended. For back compatibility, if it is used, the IntC-1 and ILC-1 support libraries are linked in. Do not write new code using this option.

Linking `os20.lib` directly is also no longer recommended. `os20.lib` does not contain any of the interrupt functions. To link legacy code `os20.lib` should be followed by the correct combination of interrupt controller and interrupt level controller libraries (see Table 18.1 and Table 18.2 above). Do not write new code using this option.

18.3 Initializing the interrupt handling support system

Before any interrupt handling routines are written the interrupt hardware needs to be configured and initialized in order that STLite/OS20 knows which hardware model is being targeted.

Both the interrupt controller and interrupt level controller have a number of configuration registers which must be correctly programmed before the peripheral can assert an interrupt signal. This also varies for each device and typically will include setting the **Mask** register to enable/disable individual interrupts (see section 18.6) and the **TriggerMode** register, see below.

The `interrupt_init_controller` function enables you to specify how the interrupt controller and interrupt level controller (if present) are configured.

```
#include <interrup.h>
void interrupt_init_controller(
void* interrupt_controller,
int interrupt_levels,
void* level_controller,
int interrupt_numbers,
int input_offset);
```

The base address and number of inputs supported by the interrupt controller and (if applicable) the interrupt level controller, on the target ST20 device must be specified. These details are device specific and can be obtained from the device datasheet.

Normally if `st20cc -runtime os20` is used when linking, then this will be performed automatically before the user’s application starts to run.

Next each interrupt level must be initialized. The `interrupt_init()` function is used to initialize a single interrupt level in the interrupt controller:

```
#include <interrup.h>
int interrupt_init(
int interrupt,
void* stack_base,
size_t stack_size,
interrupt_trigger_mode_t trigger_mode,
interrupt_flags_t flags);
```

This function enables an area of stack to be defined and also specifies the trigger mode associated with an interrupt level, that is, whether the interrupt is active when the signal is high, or low, or on a rising, falling or both edges of the signal. The stack is used to execute all interrupt handlers attached at that level so must be large enough to accommodate the largest interrupt handler.

18.3.1 Calculating stack size

The area of stack must be large enough for each interrupt handler to execute within. It must accommodate all the local variables declared within a handler and must take account of any further function calls that the handler may make. **Note:** the following:

As a general rule an interrupt handler uses the following workspace:

- Eight words of save state.
- Five words for internal pointers, for ILC-None or ILC-1 interrupt libraries, seven words for ILC-2 and eight words for ILC-3.
- Space for the user's initial stack frame (four words on an ST20-C2, three words on an ST20-C1).
- Then recursively:
  - Space for local variables declared in the function, (add up the number of words).
  - Space for calls to extra functions. For a library function allow 150 words for worst case.

For details of data representation, see the "*Implementation Details*" chapter of the "*ST20 Embedded Toolset Reference Manual*".

18.4 Attaching an interrupt handler in STLite/OS20

An interrupt handler is attached to an interrupt, using the `interrupt_install()` function:

```
#include <interrup.h>
int interrupt_install (
    int Number,
    int Level,
    void (*Handler)(void* Param),
    void* Param);
```

Once the interrupt handler is attached the interrupt should be enabled by calling `interrupt_enable` or `interrupt_enable_number` as described in section 18.6.

The function `interrupt_install_sl` enables an interrupt to be installed for use with relocatable code units. The chapter "*Building and running relocatable code*" in the "*ST20 Embedded Toolset Reference Manual*" provides details of using STLite/OS20 with relocatable code units.

18.4.1 Attaching interrupt handlers directly to peripherals

If there is no interrupt level controller on the ST20, then only one handler can be attached to each interrupt level (and the interrupt number specified to the `interrupt_install` function must be specified as -1). `interrupt_install` will then associate the specified interrupt handler with a particular interrupt level.

Example

```
#include <interrup.h>
int interrupt_stack[500];
void interrupt_handler(void* param);

int intrpt_stack[500];
void intrpt_handler(void* param);

interrupt_init(4, interrupt_stack, sizeof(interrupt_stack),
interrupt_trigger_mode_rising, 0);
interrupt_install(-1, 4, interrupt_handler, NULL);

interrupt_init(2, intrpt_stack, sizeof(intrpt_stack),
interrupt_trigger_mode_low_level, 0);
interrupt_install(-1, 2, intrpt_handler, NULL);
interrupt_enable(2);
interrupt_enable(4);

interrupt_enable_global();
```

18.4.2 Attaching interrupt handlers using an interrupt level controller

On devices which have an interrupt level controller, then multiple handlers can be attached to each level, one for each interrupt number. The act of attaching the interrupt handler at a level will result in the interrupt controller being programmed to generate the chosen interrupt level. When an interrupt occurs at an interrupt level which has multiple interrupt numbers attached, STLite/OS20 will arrange to call all the appropriate handlers for interrupts which are pending. To do this it will loop, checking for pending interrupts at the current level, until there are none outstanding. When multiple interrupt numbers are pending, the numerically highest will be called first.

Example:

```
#include <interrup.h>
int interrupt_stack[500];
void interrupt_handler(void* param);
void intrpt_handler(void* param);

interrupt_init(4, interrupt_stack, sizeof(interrupt_stack),
interrupt_trigger_mode_rising, 0);
interrupt_install(10, 4, interrupt_handler, NULL);
interrupt_install(3, 4, intrpt_handler, NULL);
/* for ILC-2 or ILC-3 type interrupt level controllers
 * interrupt_enable(4) would be replaced by
 * interrupt_enable_number(10)
 * interrupt_enable_number(3)
 */
interrupt_enable(4);
interrupt_enable_global();
```

18.4.3 Routing interrupts to external pins

ILC-3 supports the function of routing interrupts to external pins (called interrupt outputs), where additional hardware can handle the interrupt. Typically this would be used for multi-CPU systems. To set up this mode of operation, `interrupt_install` is used, and the interrupt level is specified as: (`-1` minus the number of the interrupt output). For example:

```
/* Direct interrupt number 4 to external interrupt output 2 */
interrupt_install(4, -3, NULL, NULL);
```

18.4.4 Efficient interrupt layouts

STLite/OS20 does not install the interrupt handler supplied to `interrupt_install` as the first level handler. Instead it installs its own optimized interrupt handlers to determine which interrupt number caused that interrupt level to be raised and then sets up the workspace to make C calls. STLite/OS20 picks the best code it can to minimize interrupt latency. Carefully laying out interrupts can assist this.

The most efficient case is when a single interrupt number is attached to an interrupt level, there is little work to be done and every address can be precalculated by `interrupt_install`. Devices that require absolute minimum latency should be attached like this.

For the ILC-1 and ILC-2 there are no further optimizations that can be made.

ILC-3 has more than 32 interrupt numbers. The ST20 is a 32-bit processor and therefore ILC-3 registers cross the word boundary of the machine. When two interrupt numbers attached to the same level are spread across more than one word the work required to determine the source of the interrupt increases. Thus bunching interrupt numbers between word boundaries will minimize interrupt latency.

Example:

```
/* good layout (for ILC-3) */
interrupt_install(1, 1, intrpt_handler1, NULL);
interrupt_install(31, 1, intrpt_handler2, NULL);

/* poor layout (crosses word boundary) */
interrupt_install(3, 2, intrpt_handler3, NULL);
interrupt_install(33, 2, intrpt_handler4, NULL);
```

18.5 Initializing the peripheral device

Each peripheral device has its own interrupt control register(s) which must be programmed in order for the peripheral to assert an interrupt signal. This is device dependent and so will vary between devices, but will usually involve specifying which events should cause an interrupt to be raised. The example in section 18.7 shows a setup for an Asynchronous Serial Controller (ASC). It is important that these device registers are set up after the interrupt controller and interrupt level controller. (Likewise when deleting interrupts it is important that the peripheral device interrupt control register(s) are reprogrammed first, see section 18.15).

18.6 Enabling and disabling interrupts

The following two functions can be used to set or clear the global enables bit `INTERRUPT_GLOBAL_ENABLE` in the interrupt controller's Set_Mask register:

```
#include <interrup.h>
void interrupt_enable_global();
void interrupt_disable_global();
```

When the global enables bit is set then any enabled interrupt can be asserted. When the global enables bit is not set then no interrupts can be asserted regardless of whether they are individually enabled. These two functions apply to all interrupt controllers.

18.6.1 Enabling and disabling interrupts without an ILC or with ILC-1

The following two functions take an interrupt level and set or clear the corresponding bit in the interrupt controller **Set_Mask** register:

```
#include <interrup.h>
int interrupt_enable (int Level);
int interrupt_disable (int Level);
```

This can be used to enable or disable the associated interrupt level.

Although the global enables bit can be set or cleared by these functions (as `INTERRUPT_GLOBAL_ENABLE`) this use is no longer recommended. These functions return `-1` if an illegal interrupt level is passed in.

Although both functions work on all existing ST20 processors they are not guaranteed to work for future processors with ILC-2 or ILC-3 interrupt level controllers. Thus their use is only recommended for use on chips with no interrupt level controller or with ILC-1.

The following two functions are similar to those above but take a mask which contains bits to be set or cleared in the interrupt controller **Set_Mask** register depending on the operation being performed.

```
#include <interrup.h>
void interrupt_enable_mask  (int Mask);
void interrupt_disable_mask (int Mask);
```

Like the previous functions the global enables bit can be set or cleared using the mask functions (as `1 << INTERRUPT_GLOBAL_ENABLE`) and again it is no longer recommended. Similarly these functions are only recommended for use on chips with no interrupt level controller or with ILC-1.

18.6.2 Enabling and disabling interrupts with ILC-2 or ILC-3

The following two functions apply only to ILC-2 or ILC-3 interrupt level controllers and are used to enable and disable interrupt numbers.

```
#include <interrup.h>
int interrupt_enable_number (int Number);
int interrupt_disable_number (int Number);
```

These functions allow specific interrupt numbers to be enabled and disabled independently by writing to the interrupt level controllers **Enable** registers.

18.7 Example: setting an interrupt for an ASC

This example shows how an interrupt could be set for an Asynchronous Serial Controller on an STi5500 device, this device has an ILC-1 type interrupt level controller. The example demonstrates the steps described in the previous sections to:

- Initialize the interrupt controller, see section 18.3.
- Attach an interrupt handler, see section 18.4.
- Program the peripheral device registers, see section 18.5.
- Enable an interrupt, see section 18.6.

```
#define INTERRUPT_NUMBERS 18
#define INTERRUPT_INPUT_OFFSET 18
#define INTERRUPT_CONTROLLER 0x20000000
#define INTERRUPT_LEVEL_CONTROLLER 0x20011000
#define ASC0_INTERRUPT_NUMBER 9
#define ASC_INTERRUPT_LEVEL 5

typedef struct {
  int asc_BaudRate;
  int asc_TxBuffer;
  int asc_RxBuffer;
  int asc_Control;
  int asc_IntEnables;
  int asc_Status;
} asc_t;

volatile asc_t* asc0 = (asc_t*)0x20003000;

#define ASC_MODE_8D      0x01
#define ASC_STOP_1_0     0x08
#define ASC_RUN          0x80
#define ASC_RXEN         0x100

#define ASC_BAUD_9600    (40000000 / (16*9600))

#define ASC_RX_BUF_FULL  1

interrupt_init_controller((void*)INTERRUPT_CONTROLLER, 8,
        (void*)INTERRUPT_LEVEL_CONTROLLER,
        INTERRUPT_NUMBERS, INTERRUPT_INPUT_OFFSET);

interrupt_init(ASC_INTERRUPT_LEVEL, ser_stack, sizeof(ser_stack),
     interrupt_trigger_mode_high_level, 0);

interrupt_enable_global();

if (interrupt_install(ASC0_INTERRUPT_NUMBER, ASC_INTERRUPT_LEVEL,
                      ser_handler, NULL) == 0) {
  asc->asc_Control    = ASC_MODE_8D | ASC_STOP_1_0 | ASC_RUN | ASC_RXEN;
  asc->asc_BaudRate   = ASC_BASE_9600;
  asc->asc_intEnables = ASC_RX_BUF_FUL;
  interrupt_enable(ASC_INTERRUPT_LEVEL);
}
```

If this example were transferred to a device with an ILC-2 or ILC-3 interrupt level controller the call to `interrupt_enable` (last line) would become:-

```
interrupt_enable_number(ASC0_INTERRUPT_NUMBER);
```

In section 18.15 an example is given of how to remove this interrupt.

18.8 Locking out interrupts

All interrupts to the CPU can be globally disabled or re-enabled using the following two commands:

```
#include <interrupt.h>
void interrupt_lock (void);
void interrupt_unlock (void);
```

These functions should always be called as a pair and will prevent any interrupts from the interrupt controller having any effect on the currently executing task while the lock is in place. These functions can be used to create a critical region in which the task cannot be preempted by any other task or interrupt. Calls to `interrupt_lock()` can be nested, and the lock will not be released until an equal number of calls to `interrupt_unlock()` have been made.

Note: that locking out interrupts is slightly different from disabling an interrupt. Interrupts are locked by changing the ST20's **Enables** register, which causes the CPU to ignore the interrupt controller (and any other external device), while disabling an interrupt modifies the interrupt controller's **Mask** register, and so can be used much more selectively. On the ST20-C2 locking interrupts also prevents high priority processes from interrupting and disable channels and timers.

A task must not deschedule with interrupts locked, as this can cause the scheduler to fail. When interrupts are locked calling any function that may not be called by an interrupt service routine is illegal.

18.9 Raising interrupts

The following functions can be used to force an interrupt to occur:

```
#include "interrup.h"
int interrupt_raise (int Level);
int interrupt_raise_number (int Number);
```

The first function will raise the specified interrupt level and should be used when peripherals are attached directly to the interrupt controller. The second function will raise the specified interrupt number and is for use when an interrupt level controller is present.

Note that neither function should be used to raise level sensitive interrupts. They will be immediately cleared by the interrupt hardware.

18.10 Retrieving details of pending interrupts

The following functions will return details of pending interrupts:

```
#include <interrupt.h>
int interrupt_pending (void);
int interrupt_pending_number (void);
int interrupt_test_number (int Number);
```

The first function returns which interrupt levels are pending, that is, those interrupts which have been set by a peripheral, but their interrupt handlers have not yet run. This function should be used when peripherals are attached directly to the interrupt controller. The second function returns which interrupt numbers are pending, that is, all the interrupts which are currently set by peripherals. The ST20 C compiler treats `int` as a 32-bit quantity, thus `interrupt_pending_number` can not be used on ILC-3 type interrupt level controllers because they have too large a quantity of interrupt numbers.

The final function can be used to test if any one specific interrupt number is pending. This function applies to any interrupt level controller because it does not return a mask.

18.11 Clearing pending interrupts

The following functions can be used to prevent a raised interrupt signal from causing an interrupt event to occur:

```
#include “interrupt.h”
int interrupt_clear (int level);
int interrupt_clear_number (int Number);
```

The first function clears the specified pending interrupt level and should be used when peripherals are attached directly to the interrupt controller. The second function will clear the specified interrupt number and is for use when an interrupt level controller is present. If the specified number is the only pending interrupt number attached to the interrupt level then the pending interrupt level is also cleared.

On ILC-1 only interrupts asserted in software by `interrupt_raise_number` can be cleared in this way.

18.12 Changing trigger modes

This section applies only to ST20 variants with ILC-2 or ILC-3. On these devices the following function can be used to change a specific interrupt number’s trigger mode.

```
#include <interrup.h>
int interrupt_trigger_mode_number(
int Number,
interrupt_trigger_mode_t trigger_mode);
```

When `interrupt_init` is called the user supplies a default trigger mode for all interrupt numbers attached to that interrupt level. When an interrupt is installed then the trigger mode will be set to this default. `interrupt_trigger_mode_number` can be used to change away from the default behavior set by `interrupt_init`.

18.13 Low power modes and interrupts

This section applies only to ST20 variants with ILC-2 or ILC-3. On these devices the following function can be used to configure which external interrupts can wake the ST20 from low power mode.

```
#include <interrup.h>
int interrupt_wakeup_number(
int Number,
interrupt_trigger_mode_t trigger_mode);
```

Once the ST20 has been placed in low power mode the device can be woken either when its real-time wake-up alarm triggers or when an external interrupt request is asserted. The external request is active high or active low, it cannot be edge triggered.

Note that on some ST20 variants not all external interrupt pins can be used to wake the device from low power mode, exact details can be found from the appropriate device datasheet.

18.14 Obtaining information about interrupts

The following two functions can be used to obtain interrupt state information:

```
#include <interrup.h>
int interrupt_status(int Level, interrupt_status_t* Status,
interrupt_status_flags_t flags);

int interrupt_status_number(
int Number,
interrupt_status_number_t* status,
interrupt_status_number_flags_t flags);
```

The first function provides information about the state of an interrupt level. This includes the number of interrupt handlers attached to this level and the current state of the interrupt stack, specifically the stack’s base, size and peak usage.

The second function provides information about the state of an interrupt number. For standard STLite/OS20 kernels this includes only the interrupt level to which this interrupt number is attached.

On the debug kernel `interrupt_status` and `interrupt_status_number` provide extra timing information. This extra data is not available on the deployment kernel because it would decrease interrupt performance. See the chapter “*Advanced configuration of STLite/OS20*” in the "*ST20 Embedded Toolset Reference Manual*" for further details.

18.15 Uninstalling interrupt handlers and deleting interrupts

The following function can be used to uninstall an interrupt handler:

```
#include <interrup.h>
int interrupt_uninstall(
int Number,
int Level);
```

Before `interrupt_uninstall` is used, the interrupt must be disabled on the actual peripheral device by programming the peripheral's interrupt control register(s) and then using one of the functions:

```
interrupt_disable
interrupt_disable_mask
interrupt_disable_number
```

A replacement trap handler may then be swapped in using `interrupt_install` or the interrupt may be deleted, using `interrupt_delete` if it is no longer required. If a replacement trap handler is installed the interrupt must be re-enabled on the peripheral device by programming its interrupt control register(s).

The following function will delete an initialized interrupt, allowing the interrupt level's stack to be freed:

```
#include <interrup.h>
int interrupt_delete(int Level);
```

The interrupt must be disabled by programming the peripheral's interrupt control register(s) and uninstalled by calling `interrupt_uninstall` before `interrupt_delete` is called.

Example

This example demonstrates how to delete the interrupt set up by the example given in section 18.7.

```
asc->asc_intEnables = 0;
interrupt_disable(ASC_INTERRUPT_LEVEL);
interrupt_uninstall(ASC0_INTERRUPT_NUMBER, ASC_INTERRUPT_LEVEL);
interrupt_delete(ASC_INTERRUPT_LEVEL);
```

18.16 Restrictions on interrupt handlers

Certain restrictions must be kept in mind when using interrupts on the ST20. These restrictions, and their ramifications for C are as follows:

- Descheduling and timeslicing are automatically disabled for interrupt handlers.

  Channel communications (on the ST20-C2) and any other descheduling operation are not permitted.

- Interrupt handlers must not use 2D block move instructions unless the existing block move state is explicitly saved and restored by the handler.

- Interrupt handlers must not cause traps.

  Care should be taken here to make sure that instruction sequences are not generated which could cause errors and therefore a trap to be taken. It is suggested that in the simplest case an interrupt handler should disable all traps.

18.17 Interrupt header file: interrup.h

All the definitions related to interrupts are in the single header file, `interrup.h`, see Table 18.4.

Function	Description	ILC library
`interrupt_clear`	Clear a pending interrupt.	ILC-None, ILC-1
`interrupt_clear_number`	Clear a pending interrupt number.	ILC-1, ILC-2, ILC-3
`interrupt_delete`	Delete an interrupt level.	ILC-None, ILC-1, ILC-2, ILC-3
`interrupt_disable`	Disable an interrupt level.	ILC-None, ILC-1
`interrupt_diable_global`	Global disable interrupts.	ILC-None, ILC-1, ILC-2, ILC-3
`interrupt_disable_mask`	Disable one or more interrupts.	ILC-None, ILC-1
`interrupt_disable_number`	Disable an interrupt number.	ILC-2, ILC-3
`interrupt_enable`	Enable an interrupt level.	ILC-None, ILC-1
`interrupt_enable_global`	Globally enable interrupts.	ILC-None, ILC-1, ILC-2, ILC-3
`interrupt_enable_mask`	Enable one or more interrupts.	ILC-None, ILC-1
`interrupt_enable_number`	Enable an interrupt number.	ILC-2, ILC-3
`interrupt_init`	Initialize an interrupt level.	ILC-None, ILC-1, ILC-2, ILC-3
`interrupt_init_controller`	Initialize the interrupt controller.	ILC-None, ILC-1, ILC-2, ILC-3
`interrupt_install`	Install an interrupt handler.	ILC-None, ILC-1, ILC-2, ILC-3
`interrupt_install_sl`	Install an interrupt handler and specify a static link.	ILC-None, ILC-1, ILC-2, ILC-3
`interrupt_lock`	Lock all interrupts.	ILC-None, ILC-1, ILC-2, ILC-3
`interrupt_pending`	Return pending interrupt levels.	ILC-None, ILC-1
`interrupt_pending_number`	Return pending interrupt numbers.	ILC-None, ILC-1, ILC-2
`interrupt_raise`	Raise an interrupt level.	ILC-None, ILC-1
`interrupt_raise_number`	Raise an interrupt number.	ILC-1, ILC-2, ILC-3
`interrupt_status`	Report the status of an interrupt level.	ILC-None, ILC-1, ILC-2, ILC-3
`interrupt_status_number`	Report the status of an interrupt number.	ILC-1, ILC-2, ILC-3
`interrupt_test_number`	Test whether an interrupt number is pending.	ILC-1, ILC-2, ILC-3

Table 18.4 Functions defined in `interrup.h`

Function	Description	ILC library
`interrupt_trigger_mode_number`	Change the trigger mode of an interrupt number.	ILC-2, ILC-3
`interrupt_uninstall`	Uninstall an interrupt handler.	ILC-None, ILC-1, ILC-2, ILC-3
`interrupt_unlock`	Unlock all interrupts.	ILC-None, ILC-1, ILC-2, ILC-3
`interrupt_wakeup_number`	Set wakeup status of an interrupt number.	ILC-2, ILC-3

Table 18.4 Functions defined in `interrup.h`

The full ILC library names are given in Table 18.2.

Types and macros defined to support interrupts are listed in Table 18.5 and Table 18.6.

Types	Description
`interrupt_flags_t`	Additional flags for `interrupt_init`.
`interrupt_status_t`	Structure describing the status of an interrupt level.
`interrupt_status_flags_t`	Additional flags for `interrupt_status`.
`interrupt_status_number_t`	Structure describing the status of an interrupt number.
`interrupt_status_number_flags_t`	Additional flags for `interrupt_status_number`.
`interrupt_trigger_mode_t`	Interrupt trigger modes (used in `interrupt_init`).

Table 18.5 Types defined in `interrup.h`

Macro	Description
`INTERRUPT_GLOBAL_ENABLE`	Global interrupt enables bit number

Table 18.6 Macros defined in `interrup.h`

19 Device information

Two functions are provided to return information about the ST20 family of devices. device_id returns the ID of the current device. device_name takes a device ID as input and returns a brief description of the device.

Device Identifiers are defined by the IEEE1149.1 (JTAG) Boundary-Scan Standard. This is a 32 bit number composed of a number of fields. STLite/OS20 defines a type to describe this, device_id_t. This is a union with three fields:

- id which allows the code to be manipulated as a 32 bit quantity.
- jtag which views the value as defined by the JTAG standard.
- st which views the value as used by STMicroelectronics. This divides the device code into a family and device code.

jtag and st are structs of bit-fields, which allows the elements to be accessed symbolically.

The identification code is made up as in Table 19.1.

bits	jtag	st	meaning
31-28	revision	revision	Mask revision
27-22	device_code	family	20_{10} – STAR family
21-12		device_code	Device code
11-1	manufacturer	manufacturer	32_{10} – STMicroelectronics
0	JTAG_bit	JTAG_bit	1 – fixed by JTAG

Table 19.1 Composition of identification code

19.1 Device ID header file: device.h

All the definitions related to device identification are in the single header file, device.h, see Table 19.2.

Function	Description
device_id	Returns the ID of the current device.
device_name	Returns the name of the current device.

Table 19.2 Functions defined in device.h

Types	Description
device_id_t	Device ID.

Table 19.3 Types defined in device.h

20 Caches

Cache provides a way to reduce the time taken for the CPU to access memory and so can greatly increase system performance.

20.1 Introduction

All ST20 processors that support cache use similar hardware and the operation of the caches is the same, however, the blocks of memory that can be cached vary between ST20 devices, see the appropriate device datasheets for details.

The ST20 cache system provides a read-only instruction cache and a write-back data cache.

There is a risk when using cache that the cache can become *incoherent* with main memory, meaning that the contents of the cache conflicts with the contents of main memory. For example, devices that perform *direct memory access* (DMA) modify the main memory without updating the cache, leaving its contents invalid. For this reason enabling the data cache for blocks of memory accessed by the DMA engine is not recommended. Note that on an ST20-C2 core, device access instructions (generated with `#pragma ST_device`) bypass the cache and can be used to solve some cache coherency issues.

20.1.1 Data caches with internal SRAM

Some ST20 devices have a data cache which must be reserved by the linker in order to prevent it from being corrupted by the application. This is described in section 5.2.

20.2 Initializing the cache support system

Before any call is made to the cache handling routines the cache control hardware needs to be configured and initialized in order that STLite/OS20 knows which hardware model is being targeted.

If the `st20cc -runtime os20` command is used when linking, then the cache controller will be configured automatically before the user's application starts to run.

If `st20cc -runtime os20` is not used the `cache_init_controller` function enables you to specify how the cache control hardware is configured:

```
#include <cache.h>
void cache_init_controller(
void* cache_controller,
cache_map_data_t cache_map);
```

Both the cache controller address and the cache map are device specific. The cache controller address can be obtained from the device datasheet. The correct cache map can be found in the `cache_init_controller` function definition, see “*Part 4 - STLite/OS20 functions*” in the "*ST20 Embedded Toolset Reference Manual*".

Note: some ST20 devices have two base addresses one for the instruction cache and one for the data cache, for example the STi5514 and the STV0396. In this case the base address that should be passed to the `cache_init_controller` function is the numerically smaller of these addresses.

20.3 Configuring the caches

On any ST20 device with a data cache the `cache_config_data` function is used to configure the data cache to treat certain blocks of memory as cacheable or non-cacheable. Note that by default all configurable blocks are set to non-cacheable therefore for all devices with a data cache the use of the `cache_config_data` function is vital to achieve maximum performance.

```
#include <cache.h>
int cache_config_data(
  void* start_address,
  void* end_address,
  cache_config_flags_t flags);
```

There are two types of ST20 instruction cache, configurable and fixed. A fixed instruction cache can only be enabled or disabled, it cannot be selectively applied to specific blocks of memory.

On devices which have a configurable instruction cache the function `cache_config_instruction` is used to enable or disable specific blocks of memory. A configurable instruction cache, like the data cache, will by default treat all configurable blocks as non-cacheable. For devices with a configurable instruction cache the use of `cache_config_instruction` is necessary to achieve maximum performance.

```
#include <cache.h>
int cache_config_instruction(
  void* start_address,
  void* end_address,
  cache_config_flags_t flags);
```

20.4 Enabling and disabling the caches

The caches are enabled using the following two functions:

```
#include <cache.h>
int cache_enable_data();
int cache_enable_instruction();
```

The first function invalidates the data cache (see section 20.7), before writing to the **EnableDCache** register thereby enabling the data cache. The second function is similar but operates on the **EnableICache** register.

If the target application requires the caches to be disabled at some later point the following two functions can be used.

```
#include <cache.h>
int cache_disable_data();
int cache_disble_instruction();
```

Disabling the cache can potentially take a long time to complete, during this time the processor will be unable to handle interrupts or perform any other time critical task.

20.5 Locking the cache configuration

The cache can be locked using the following function:

```
int cache_lock();
```

It is recommended that all cache configuration is performed at boot time and then never modified. To prevent accidental modification ST20 devices can lock the cache configuration preventing it from being changed until the hardware is reset.

20.6 Example: setting up the caches

This example shows how the caches could be set up for STi5510 devices. This example demonstrates the steps described in the previous sections to:

- Initialize and configure the cache hardware, see section 20.2.
- Enable the data and instruction caches, see section 20.4.
- Lock the cache configuration, see section 20.5.

This example uses the header file `<chip/STi5510addr.h>` supplied in the ST20 Embedded Toolset's standard configuration files directory: `$ST20ROOT/include`. The header file contains the base address of the cache controller, defined as `CacheControlAddr`.

```
#include <chip/STi5510addr.h>
#include <cache.h>

cache_init_controller((void*) CacheControlAddr, cache_map_sti5510);

cache_enable_instruction();

/* cache all possible memory */
cache_config_data(0x80000000, 0x7fffffff, cache_config_enable);

/* except region required for DMA */
cache_config_data(0x40010000, 0x4001ffff, cache_config_disable);

cache_enable_data();
cache_lock();
```

20.7 Flushing and invalidating caches

When the cache is enabled any data written to main memory will be stored in the cache and marked as *dirty* so that at some point in the future it can be properly stored to main memory. A cache flush causes all dirty cache lines to be written immediately to main memory.

Invalidating a cache causes the cache to forget its entire contents thus forcing it to reload all data from main memory.

Note: that on ST20 devices, flushing the cache will also cause it to be invalidated. After a cache flush all data will be reloaded from main memory.

In some applications it is useful to force a cache flush or invalidate, this can be achieved using the following three functions:

```
int cache_flush_data(void* reserved1, void* reserved2);
int cache_invalidate_data(void* reserved1, void* reserved2);
int cache_invalidate_instruction(void* reserved1, void* reserved2);
```

Each of these functions takes two arguments that are reserved for future use by STLite/OS20, users must supply NULL as each argument.

20.7.1 Relocatable code units

When caches are enabled extra care must be taken when handling relocatable code units. The advise given in the chapter “*Building and running relocatable code*” in the "*ST20 Embedded Toolset Reference Manual*" should be followed to ensure cache coherency is maintained.

20.8 Cache header file: cache.h

All the definitions related to the caches are in the single header file, `cache.h`, see Table 20.1.

Function	Description
`cache_config_data`	Configure the data cache.
`cache_config_instruction`	Configure the instruction cache.
`cache_disable_data`	Disable the data cache.
`cache_disable_instruction`	Disable the instruction cache.
`cache_enable_data`	Enable the data cache.
`cache_enable_instruction`	Enable the instruction cache.
`cache_flush_data`	Flush the data cache.
`cache_init_controller`	Initialize the cache controller.
`cache_invalidate_data`	Invalidate the data cache.
`cache_invalidate_instruction`	Invalidate the instruction cache.
`cache_lock`	Lock the cache configuration.
`cache_status`	Report the cache status

Table 20.1 Functions defined in `cache.h`

The types defined to support the cache API are listed in Table 20.2.

Types	Description
`cache_config_flags_t`	Additional flags for `cache_config_data`.
`cache_map_data_t`	Description of cacheable memory available on a particular ST20 variant (used by `cache_init_controller`).
`cache_status_t`	Structure describing the status of the cache.

Table 20.2 Types defined in `cache.h`

21 ST20-C1 specific features

STLite/OS20 has many features, some of which depend on a timer peripheral being present, for example, functions such as `semaphore_wait_timeout` and `time_now`.

The ST20-C1 core does not have a built-in timer peripheral. In order, for the ST20-C1 version of STLite/OS20 to provide the full API, you will need to incorporate a timer plug-in module into any applications built for the ST20-C1 cores. The plug-in module is board specific, but the interface is generic enough to allow STLite/OS20 to take advantage of any timer peripherals that are present.

STLite/OS20 can be used with or without the plug-in module, however, when accessing timer related functions without a plug-in module present, a run-time error occurs so take care when not using the plug-in module.

Internally STLite/OS20 uses a standardized low level timer API which accesses functions provided by the plug-in module via function pointers, see Figure 21.1. This is so that the application can be built with or without the plug-in module. Linkage between STLite/OS20 and the plug-in module is performed at run-time as opposed to compile-time so that the only change needed to the application is an additional call to the plug-in module's initialization function.

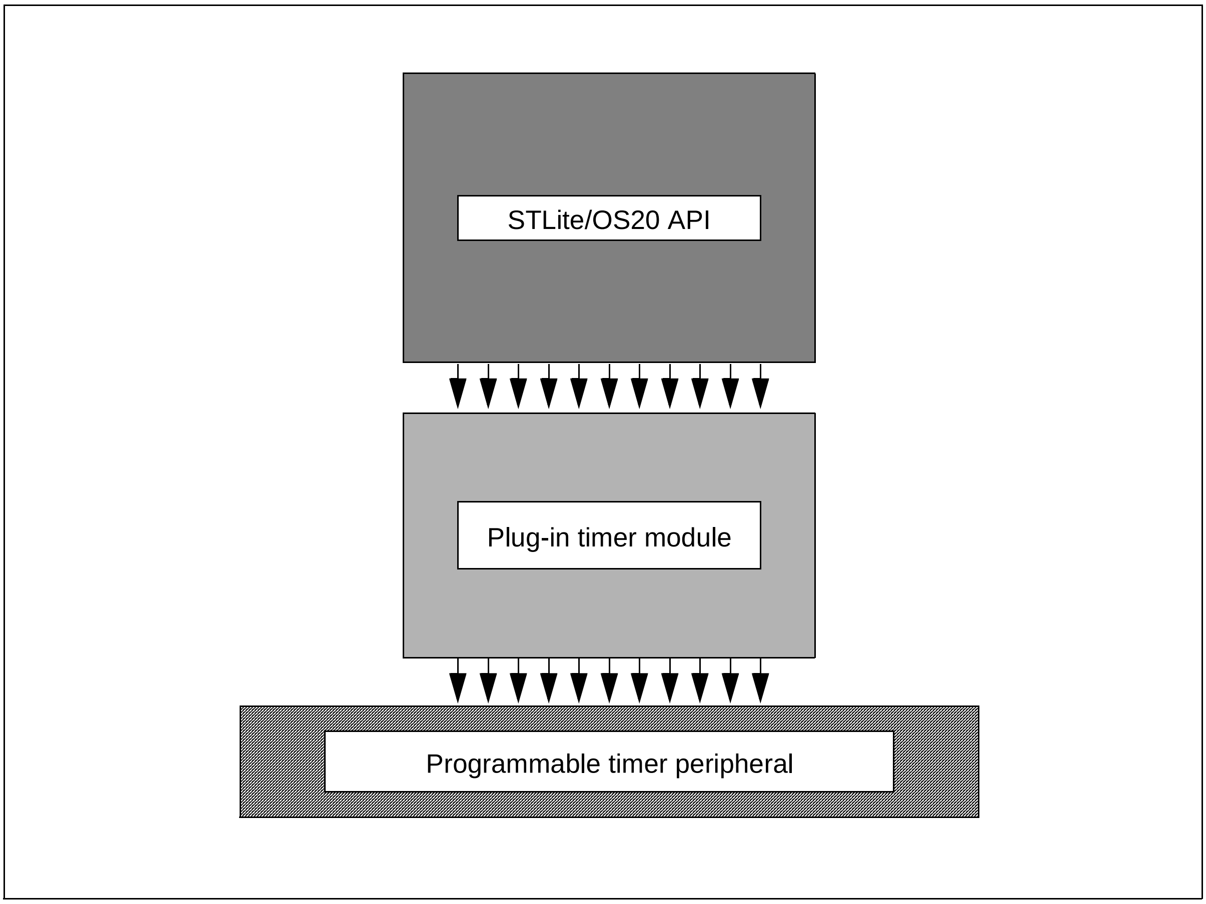


Figure 21.1 Plug-in timer model for the ST20-C1

The plug-in module must provide an initialization function which the application can call. Upon calling the initialization function, the module will initialize the programmable timer and pass a structure detailing all of the functions' locations into STLite/OS20 via a function called `timer_initialize`. This is a STLite/OS20 ST20-C1 specific function call. At this stage the plug-in module is linked into the STLite/OS kernel.

The syntax of the timer API is consistent with the remainder of STLite/OS20. The naming convention has an object oriented approach:

<class>_<type_of_operation>

Function	Description
`timer_read`	Read the timer.
`timer_set`	Set the timer.
`timer_enable_int`	Enable the timer interrupt
`timer_disable_int`	Disable the timer interrupt
`timer_raise_int`	Raise a timer interrupt

Table 21.1 Internal STLite/OS20 Timer API

Before the plug-in module is initialized, the STLite/OS20 kernel must be initialized and started by calling `kernel_initialize` and `kernel_start`, additionally the interrupt controller must be initialized by calling `interrupt_init_controller`.

Finally note that because the timer module is outside of STLite/OS20 the number of ticks per second is defined by the plug-in timer module. This has the advantage that the ticks per second can be tailored for the specific application, a short tick when high accuracy is required, a longer tick when long durations need to be timed. Note that this means care must be taken when porting code to a different device as the number of ticks per second is likely to change.

21.1 ST20-C1 example plug-in timer module

A plug-in module is provided as example code for the ST20MC2 and ST20DC1 evaluation boards. This can be found in the `examples/os20/c1timer` directory. The readme file supplied with the example explains how to build and run the example.

This example contains two completely separate timer modules `dc1timer.c` and `mc2timer.c`. These can each be used standalone if required (that is only one need be linked to your application). However, the supplied example has a single timer initialize function called `c1_timer_initialize` that uses `device_id` to determine which timer module to use.

21.1.1 PWM peripheral

Both timer plug-in modules use the on-chip PWM peripheral to provide the timer functionality. This peripheral is described here in sufficient detail to explain how the example works.

The PWM peripheral has a programmable timer which you program to cause an interrupt at a specified time.

The **CaptureCount** register is a 32-bit counter that is incremented regularly. The **Compare** register is set by your application. When the value in the **CaptureCount** register becomes equal to the value in the **Compare** register an interrupt is generated.

Table 21.2 provides a list of registers which are actively used by the plug-in module.

Register	Description
Control	Used to initialize PWM peripheral.
InterruptEnable	Enable and disable interrupts by this register.
CaptureCount	32-bit counter.
Compare	Time at which an event should occur.

Table 21.2 PWM registers used by the plug-in module.

Control

The **Control** register controls the top level function of the PWM peripheral. In particular it contains a **Capture** enable bit that causes the **CaptureCount** register to start counting and a **Capture** prescale value which controls the rate at which the **CaptureCount** register runs. By default, the prescale value is set to 0.

InterruptEnable

The **InterruptEnable** controls which events will cause an interrupt to be asserted. In particular the register contains a bit which when set will cause an interrupt to be asserted then the **CaptureCount** register becoming equal to the **Compare** register.

CaptureCount

The **CaptureCount** register is a 32-bit counter that is clocked by the system clock. The counter can be prescaled by the **Capture** prescale value stored in the **Control** register.

Compare

The **Compare** register contains the time which is compared against the **CaptureCount** register. When these are equal the timer will request an interrupt depending on the state of the **InterruptEnable** register.

21.2 Plug-in timer module header file: c1timer.h

All the definitions related to ST20-C1 plug-in timer modules are defined in a single header file, `c1timer.h`, see Table 21.3.

Function	Description
`timer_initialize`	Initialize the timer plug-in module
`timer_interrupt`	Notify STLite/OS20 that the timer has expired.

Table 21.3 Functions defined in `c1timer.h`

Types	Description
`timer_api_t`	Set of function pointers to be used as a plug-in timer module.

22 ST20-C2 specific features

Additional features:

- Channels

  The ST20-C2 support a point-to-point unidirectional communications channel, which can be used for communication between tasks on the same processor, and with hardware peripherals on the ST20.

- High priority processes

  High priority processes run outside of the normal STLite/OS20 scheduling regime, using the ST20's hardware scheduler. A high priority process is created using the `task_create` or `task_init` functions and specifying the `task_flags_high_priority_process` flag. High priority processes will always pre-empt normal STLite/OS20 tasks (irrespective of the task's priority), and as this takes advantage of the ST20's hardware scheduler, high priority processes can respond faster than a normal STLite/OS20 task.

  In general, high priority processes should be regarded as the equivalent of interrupt handlers for those peripherals which have a channel style interface.

  However, because high priority processes run outside of the STLite/OS20 scheduling regime, they only have very limited access to STLite/OS20 library functions. In general they can only call functions which are implemented directly in hardware, in particular this means they can only use channels and FIFO based semaphores, not priority based semaphores or message queues.

- Two dimensional block move

  A number of instructions are provided which allow two dimensional blocks or memory to be moved efficiently. This is especially useful in graphical applications.

22.1 Channels

STLite/OS20 supports the use of channels by all tasks (both normal, that is, low and high priority).

Channels are a way of transferring data from one task to another, and they also provide a way of synchronizing the actions of tasks. If one task needs to wait for another to reach a particular state, then a channel is a suitable way of ensuring that happens.

If one task is sending and one receiving on the same channel then whichever tries to communicate first will wait until the other communicates. Then the data will be copied from the memory of the sending task to the memory of the receiving task and both tasks will then continue. If only one task attempts to communicate then it will wait forever.

A channel communicates in one direction, so if two tasks need bidirectional communication, then two channels are needed, one in each direction. Any data can be passed down a channel, but the user must ensure that the tasks agree a protocol in order to interpret the data correctly.

It is the responsibility of the programmer to ensure that:

- data sent by one task is received by another;
- there is never more than one task sending on one channel;
- there is never more than one task receiving on one channel;
- the amount of data sent and received are the same;
- the type of data sent and received are the same.

If any of these rules are broken then the effect is not defined.

Channels between tasks are created by using the data structure `chan_t` and initializing it by calling a library function. Channel input and output functions are then used to pass data. Separate functions exist for input and output and the two must be paired for communication between two tasks to take place. The header file `chan.h` declares the `chan_t` data type and channel library functions.

If one task has exclusive access to a particular resource and acts as a server for the other tasks, then channels can also act as a queuing mechanism for the server to wait for the next of several possible inputs and handle them in turn.

A channel used to communicate between two tasks on the same processor is known as a ‘*soft channel*’. A channel used to communicate with a hardware peripheral is known as a ‘*hard channel*’.

When the STLite/OS20 scheduler is enabled (by calling `kernel_start`), channel communication will result in traps to the kernel, which will ensure that correct scheduling semantics are maintained.

22.1.1 Creating a channel

STLite/OS20 refers to channels using a `chan_t` structure. This needs to be initialized before it can be used, by using one of the following functions:

```
chan_t *chan_create(void)
chan_t *chan_create_address(void *address)
chan_init(chan_t *chan);
void chan_init_address(chan_t *chan, void *address);
```

The `_create` versions allocate memory for the data structure from the system partition and initialize the channel to their default state. `chan_create` creates a '*soft*' channel, `chan_create_address` creates a '*hard*' channel.

The `_init` versions also initialize a channel, but the allocation of memory for `chan_t` is left to the user. `chan_init` initializes a '*soft*' channel and `chan_init_address` initializes a '*hard*' channel:

For example:

```
#include <chan.h>
/* Initialize a soft channel */
chan_t soft_chan;
chan_init(&soft_chan);

/* Initialize a hard channel to link 0 input channel */
chan_t chan0;
chan_init_address(&chan0, (void*)0x80000010);
```

22.1.2 Communications over channels

Once a channel has been initialized, there are several functions available for communications:

```
void chan_in(chan_t *chan, void* cp, int count);
void chan_out(chan_t *chan, const void* cp, int count);

int chan_in_int(chan_t *chan);
void chan_out_int(chan_t *chan, int data);

char chan_in_char(chan_t *chan);
void chan_out_char(chan_t *chan, char data);
```

These functions will transfer a block of data (`chan_in` and `chan_out`), an integer (`chan_in_int` and `chan_out_int`) or a character (`chan_in_char` and `chan_out_char`).

Each call of one these functions represents a single communication. The task will not continue until the transfer is complete.

Care needs to be taken to ensure that data is only transferred in one direction across the channel, and that the sending and receiving data is the same length, as this is not checked for at run time.

For example, the following code will use channel my_chan to send a character followed by an integer followed by a string:

```
#include <chan.h>
char ch1;
int n1;

chan_out_char (my_chan, ch1);
chan_out_int (my_chan, n1);
chan_out (my_chan, "Hello", 5);
```

To receive this data on channel my_chan, the following code could be used:

```
#include <chan.h>
char ch, buffer[5];
int n;

ch = chan_in_char (my_chan);
n = chan_in_int (my_chan);
chan_in (my_chan, buffer, 5);
```

22.1.3 Reading from several channels

There are many cases where a receiving task needs to listen to several channels and wishes to detect which one has data ready first. The ST20-C2 micro-kernel provides a mechanism to handle this situation called an alternative input. This is implemented in STLite/OS20 by the following function:

```
int chan_alt(chan_t ** chanlist, int nchans,
    const clock_t *timeout);
```

chan_alt takes as parameters an array of channel pointers, and a count of the number of elements in the array. It returns the index of the selected channel, starting at zero for the first channel. The selected channel may then be read, using the input functions described above in section 22.1.2. Any channels that become ready and are not read will continue to wait. In addition an optional timeout may be provided, which allows chan_alt to be used in a polling mode, or wait until a specified time before returning, whether a channel has become ready for reading or not. Timeouts for channels are implemented using hardware and so do not increase the application's code size.

Normally chan_alt is used with the time-out value TIMEOUT_INFINITY, in which case only one of the channels becoming ready (that is, one of the sending tasks that is trying to send) will cause it to return. When one or more channels are ready then one will be selected. If no channel becomes ready then the function will wait for ever. **Note**: that the header file ostime.h must be included when using this function.

To input from an array of channels, the returned index can be used as an index into the channel array, for example:

```
#include <chan.h>
#include <ostime.h>
#define NUM_CHANS 5

chan_t *data_chan[NUM_CHANS];

int selected, x;

...

selected = chan_alt(data_chan, NUM_CHANS, TIMEOUT_INFINITY);
x = chan_in_int(data[selected]);
deal_with_data (x, selected);
```

`chan_alt` is implemented so that it does not poll while it is waiting, but is woken by one of the input channels becoming ready. This means that the processor is free to perform other processing while the task is waiting.

When it is necessary to poll channels, this can be performed by specifying a timeout of `TIMEOUT_IMMEDIATE`. This will cause the function to perform a single poll of the channels to identify whether any channel is ready. If no channel is ready then it returns `-1`.

Polling channels is inefficient and should only be used when there is a significant interval between polls, since otherwise the processor can be occupied entirely with polling. Polling is usually only used when a task is performing some regular or ongoing task and occasionally needs to poll one or more input channels for control signals or feedback.

Finally, it is also possible to specify that `chan_alt` should only wait until a specified time before returning, even if none of the specified channel has become ready for input. If the list consists of only one channel then this becomes a time-out for a single channel input. If no channel becomes ready before the clock reaches the given time, then the function returns and the task continues execution.

When used in this way `chan_alt` returns on the occurrence of the earlier of either an input becoming ready on any of the channels or the time. The time given is an absolute time which is compared with the timer for the current priority.

The value **`-1`** is returned if the time expires with no channel becoming ready. If a channel becomes ready before the time then the index of the channel in the list (starting from 0) is returned.

For example, the following code imposes a time out of `wait` ticks when reading from a single channel `chan`:

```
#include <ostime.h>
#include <chan.h>
int time_out_time, selected, x;

time_out_time = time_plus (time_now (), wait);
selected = chan_alt (&chan, 1, &time_out_time);

switch (selected)
{
  case 0:          /* channel input successful */
    x = chan_in_int (chan);
    deal_with_data (x);
    break;
  case -1:          /* channel input timed out */
    deal_with_time_out ();
    break;
  default:
    error_handler ();
    break;
}
```

The use of timers is described in Chapter 17.

22.1.4 Deleting channels

Channels may be deleted using `channel_delete`, see the function description in “*Part 4 - STLite/OS20 functions*” in the "*ST20 Embedded Toolset Reference Manual*", for full details.

22.1.5 Channel header file: chan.h

All the definitions related to ST20-C2 channel specific functions are in the single header file, `chan.h`, see Table 22.1 and Table 22.2.

Function	Description
`chan_alt`	Waits for input on one of a number of channels
`chan_create`	Create a soft channel.
`chan_create_address`	Create a hard channel.
`Chan_delete`	Delete a channel.
`chan_in`	Input data from a channel
`chan_in_char`	Input character from a channel
`chan_in_int`	Input integer from a channel
`chan_init`	Initialize a channel
`chan_init_address`	Initialize a hardware channel
`chan_out`	Output data to a channel
`chan_out_char`	Output character to a channel
`chan_out_int`	Output integer to a channel
`chan_reset`	Reset channel.

Table 22.1 Functions defined in `chan.h`

Types	Description
`chan_t`	A channel

Table 22.2 Types defined in `chan.h`

22.2 Two dimensional block move support

Graphical applications often require the movement of two dimensional blocks of data, for example to perform windowing, overlaying. The ST20-C2 contains instructions to perform efficient copying, overlaying and clipping of graphics data based on byte sized pixels.

A two dimensional array can be implemented by storing rows adjacently in memory. Given any two two-dimensional arrays implemented in this way, the instructions provided can copy a section (a block) of one array to a specified address in the other.

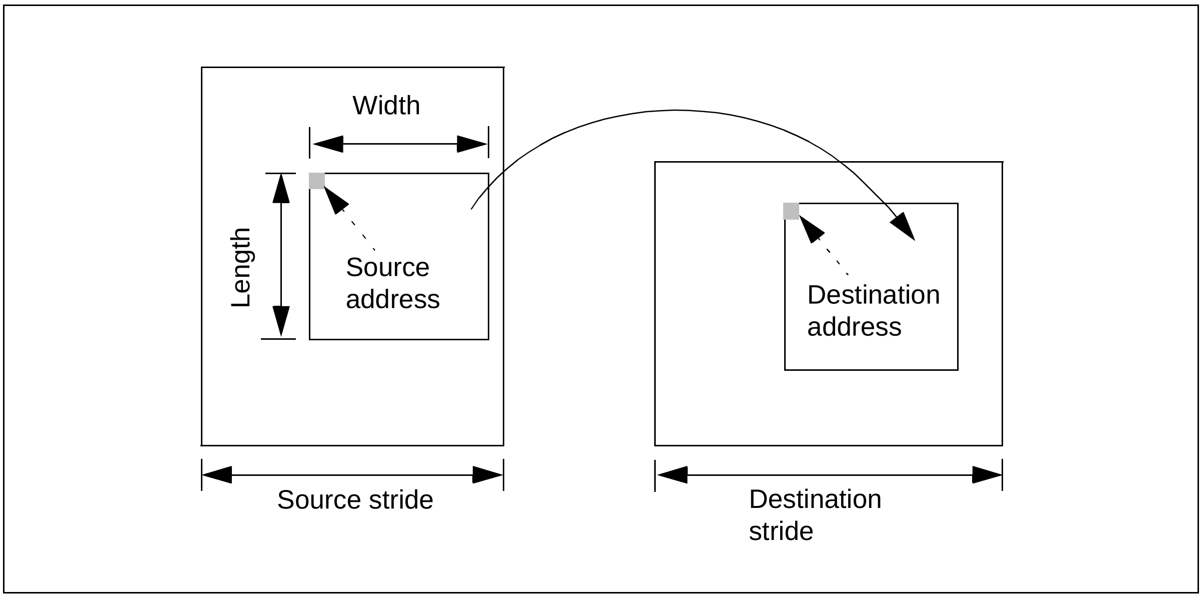


Figure 22.1 Two dimensional block move

To perform a two dimensional move, 6 parameters are required (see Figure 22.1), these are:

- The address of the first element of the source block to be copied
  – this is called the *source address*.
- The address of the first element of the destination block
  – this is called the *destination address*.
- The number of bytes in each row in the block to be copied
  – this is called the *width* of the block.
- The number of rows in the block to be copied
  – this is called the *length* of the block.
- The number of bytes in each row in the source array
  – this is called the *source stride*.
- The number of bytes in each row in the destination array
  – this is called the *destination stride*.

The two stride values are needed to allow a block to be copied from part of one array to another array where the arrays can be of differing size.

None of the two dimensional moves has any effect if either the *width* or *length* of the block to copy is equal to zero. Also a two dimensional block move only makes sense if the *source stride* and *destination stride* are both greater or equal to the *width* of the block being moved. The effect of the two dimensional moves is undefined if the source and destination blocks overlap.

Instructions are provided which allow a whole block to be moved, or only the zero or non-zero values.

STLite/OS20 provides three functions which give access to these instructions:

```
void move2d_all(const void *src, void *dst,
    int width, int nrows,
    int srcwidth, int dstwidth);
```

```
void move2d_non_zero(const void *src, void *dst,
    int width, int nrows,
    int srcwidth, int dstwidth);
```

```
void move2d_zero(const void *src, void *dst,
    int width, int nrows,
    int srcwidth, int dstwidth);
```

where:

- `move2_all` copies the whole of the block of `nrows` rows each of `length` bytes from the source to the destination.
- `move2d_non_zero` copies the non zero bytes in the block leaving the bytes in the destination corresponding to the zero bytes in the source unchanged. This can be used to overlay a non rectangular picture onto another picture.
- `move2d_zero` copies the zero bytes in the block leaving the bytes in the destination corresponding to the non zero bytes in the source unchanged. This can be used to mask out a non rectangular shape from a picture.

22.2.1 Two dimensional block move header file: move2d.h

All the definitions related to ST20-C2 two dimensional block move specific functions are in the single header file, `move2d.h`, see Table 22.3.

Function	Description	Callable from ISR/ HPP
`move2d_all`	Two dimensional block move.	HPP
`move2d_non_zero`	Two dimensional block move of non-zero bytes.	HPP
`move2d_zero`	Two dimensional block move of zero bytes.	HPP

Table 22.3 Functions defined in `move2d.h`

All functions are callable from an STLite/OS20 task or a high priority process (HPP), however, none of them can be called from an interrupt service routine.

Appendices

A Hardware breakpoint allocation

This appendix describes the rules that govern how many hardware breakpoints and data watchpoints can be setup with respect to the diagnostic controller unit (DCU) used. An overview of the facilities provided by the DCU to support debugging is given in section 7.2.4. For information describing the DCU hardware, refer to the device data sheet.

A.1 DCU2 hardware

A DCU2 contains the following breakpoint resources:

- One code breakpoint block configured with up to two single address breakpoints.
- One code breakpoint block configured with up to two single address breakpoints or as a code breakpoint count, or as a code breakpoint range block.
- One data breakpoint block.

A.1.1 DCU2 allocation rules

- Both code breakpoint blocks can support two breakpoints giving a maximum of four code breakpoints.
- For a breakpoint block to contain two code breakpoints, each breakpoint must be an enabled single address breakpoint that is not a counted breakpoint. The following `break` commands setup single address code breakpoints:

```
> break -hardwarebreakpoint func
> break -hardwarebreakpoint <main.c 24>
```

- If a code breakpoint block contains a disabled breakpoint, then that block can only contain that breakpoint.
- If a code breakpoint block contains a code breakpoint that is sequenced by or sequences another DCU2 function, then that breakpoint block can only contain that breakpoint.
- Only one counted code breakpoint can be setup.
- Only one code breakpoint over an address range can be setup.
- A counted breakpoint and a ranged breakpoint cannot be setup at the same time.
- One data breakpoint can be setup.
- If a breakpoint block contains two code breakpoints, neither breakpoint may be disabled.

A.2 DCU3 hardware

A DCU3 can be implemented with between 0-31 ‘*compare blocks*’, each configurable as either code breakpoints (*code breakpoint mode*) or as data breakpoints (*data breakpoint mode*).

The number of compare blocks a DCU3 has is dependent on its implementation. For the `STV0396` and the `STi5514`, the first devices to include DCU3, the DCU3 has been implemented with four compare blocks.

A.2.1 DCU3 allocation rules

- A compare block can be setup to support either one or two code breakpoints or a single data breakpoint function.
- A compare block in code breakpoint mode can be setup as a counted breakpoint or as a ranged breakpoint.
- In code breakpoint mode a compare block supports a maximum of two breakpoints.
- For a compare block to contain two code breakpoints, each breakpoint must be an enabled single address breakpoint that is not a counted breakpoint.
- In code breakpoint mode a compare block can only contain one breakpoint if that breakpoint is either disabled or will generate a trigger out signal.
- If a code breakpoint block contains a code breakpoint that is sequenced by or sequences another DCU3 function, then that compare block can only contain that breakpoint.
- In data breakpoint mode a compare block can contain one data breakpoint.

B Glossary

address space

In the context of the toolset debugger a memory and register state.

Areg

The register at the top of the evaluation stack. See evaluation stack.

atomic

An atomic operation is one in which no interruption can take place in between loading, modifying and storing a variable or data structure.

base of memory

The lowest address in the memory space. The ST20 has a signed address space, so the lowest address is MostNeg.

Breakpoint

See Code breakpoint.

Breg

The register in the middle of the evaluation stack. See evaluation stack.

C1

The ST20-C1 core.

C2

The ST20-C2 core.

call graph profiling

A type of execution profiling which estimates the percentage of the total run-time spent in each function and its children, and counts the number of times each function is called, and by which function. *cf.* flat profiling.

call stack

The stack of nested function calls which have not returned within the current task.

channel

A mechanism for synchronized, unbuffered communication between tasks, or with a peripheral.

Code breakpoint

A breakpoint on an instruction.

command file

A file containing a script written in command language.

command language

A language used to control the ST20 Embedded Toolset tools: `st20cc`, `st20run`, `st20sim`, `st20libr` and `stemi`.

configuration

An arrangement of hardware or software elements or both. The settings of registers to control the behavior of peripherals, or the commands to set such registers.

configuration file

A command file describing the application or target configuration.

core

The CPU silicon module ST20-C1 or ST20-C2.

core dump

A file which contains a record of the state of memory and CPU registers.

Creg

The register at the bottom of the evaluation stack. See evaluation stack.

cycle time definition file

A simulator control file, which defines the cycle times to execute each instruction and to perform memory accesses.

Data breakpoint

A breakpoint on data.

DCU

Diagnostic Controller Unit, see diagnostic controller.

debugger

A tool to aid the finding of faults in a program.

device

A silicon chip or a peripheral.

diagnostic controller

A silicon module which has access to the state of the CPU and memory, which is used for monitoring and controlling execution during debugging.

driver tool

The tool `st20cc` used to build binary code from source files and libraries.

evaluation stack

Three registers, arranged as a hardware stack, used for evaluating expressions and holding instruction arguments and results.

execution profiling

A type of profiling which estimates the percentage of the total run-time spent in each function and counts the number of times each function is executed. cf call graph profiling.

flat profiling

A type of execution profiling which estimates the percentage of the total run-time spent in each function. *cf* call graph profiling.

frame

The area of stack for a function call which has not yet returned. One level in the call stack.

function profiling

A type of profiling which measures how much each function is executed. Function profiling may be flat or produce a call graph.

hardware configuration

A description of the target hardware. This may take the form of a configuration file.

hex format

A ROM ASCII format used by the linker when generating a ROM image file suitable for blowing a ROM device.

idle profiling

A type of profiling which measures how much time the processor is idle.

Iptr

The instruction pointer, a register which holds the address of the next instruction to be executed.

JEI

A host/target interface between a JTAG port on a ST20 target board and an Ethernet network, connected to one or more UNIX, Linux or Windows hosts.

JPI

A host/target interface between a JTAG port on a ST20 target board and a PC parallel port, where the PC is running Windows.

JTAG port

A standard serial test access port, primarily for testing, but used on the ST20 for communications with the DCU.

link

A communications connection.

linked unit

A single file which has been produced by the linker and is suitable for loading onto a target device via the DCU.

lister

A tool for interpreting binary code files.

lku file

Linked unit file.

MinInt

The minimum integer, 0x80000000, which is also the base address of memory and may be used as a null pointer.

module

A section of a library containing the code from one object file which can be separately loaded.

MostNeg

The most negative integer, 0x80000000, which is also the base address of memory and may be used as a null pointer.

MostPos

The most positive integer, 0x7FFFFFFF, which is also the top address of memory.

mutex

A mutual exclusion flag used to acquire exclusive access to an object. The mutex is ‘locked’ to prevent simultaneous access to data or a process by several threads, that is, only one thread may access the protected object at a time. Once the thread has finished with the object the mutex is ‘unlocked’ and the object is made available.

OS/20

See STLite/OS20

OS-Link

A serial communications link.

process

A task. A section of code that can run in parallel to other processes.

processor

A hardware device that can execute a program.

profiling

Compiling statistics on how much each section of code is used during an application run. See flat profiling, idle profiling.

programmable engine

A hardware device with a degree of programmability that is dedicated to a particular task, for example, as an MPEG encoder or software modem.

pseudo-op

A statement in assembler code that is not a processor instruction but is a signal to control the simulator.

S-record format

A ROM ASCII format used by the linker when generating a ROM image file suitable for blowing a ROM device

sampling

A profiling technique in which the state of the CPU is tested at regular intervals.

segment of memory

A named block of memory or memory-mapped peripherals.

ST Micro Connect

A host/target interface allowing connection between an Ethernet, Parallel port or USB based host to an ST development board with debug support. Ethernet connection is supported for UNIX, Linux and Windows hosts; Parallel port and USB connection is supported for Windows hosts. See Chapter 6 or the '*ST Micro Connect datasheet*' - *ADCS 7154764* for further details.

ST20

A family of 32-bit processors that contain a number of modules, including a core and peripherals, some of which may be application-specific.

ST20-C1

A 32-bit ST20 core designed for low power applications and as an embedded programmable engine.

ST20-C2

A 32-bit ST20 core designed with on-chip multi-tasking support for real time applications. Used for example, in set-top box and digital versatile disc (DVD) applications and in digital TV.

st20cc

A tool for building ST20 binary code from ANSI C source code.

st20libr

A librarian for ST20 libraries.

st20list

A lister for ST20 binary code.

st20run

A tool for loading, running and debugging code on a ST20 target from a host.

st20sim

An ST20 simulator.

statistics tag

A pseudo-op used to turn statistics generation on or off.

STLite/OS20

A run-time system that uses the on-chip microcode support for multi-tasking.

TAP

Test Access Port.

target

A hardware processor (or possibly a simulator) that will run the application program.

task

A section of code that can run in parallel to other tasks, sometimes called a process.

test access port

A serial JTAG port, primarily for testing, but used on the ST20 for communications with the DCU.

thread

A task.

time profiling

A type of profiling which estimates how much processor time is used by each function.

timer

A clock.

trace tag

A pseudo-op used to change the level of tracing.

Watchpoint

See Data breakpoint.

Wdesc

Workspace descriptor.

word

Four bytes.

work space

Memory used for local data.

workspace descriptor

Wdesc. The ST20-C2 CPU register that holds a pointer to local data (normally to the current stack position) and the priority bit.

workspace pointer

Wptr. The pointer part of the workspace descriptor that points to local data, normally to the current stack position.

Wptr

Workspace pointer.

Index

Symbols

Numerics

A

B

C

L

M

N

O

P

T

U

V

W