ST20 Embedded Software Toolset Training

Release R2.0
Purpose and Objectives

Purpose

This 2 day training course is designed to familiarise the learner with …

• The ST20 Toolset
• The Development Environment
• OS20 Real Time Operating System

Training Objectives

At the end of this training you will be able to utilise the ST20 Toolset including OS20 in the development of your application code.
Methods employed in this course are …

- Trainer Presentations
- Structured Exercises
- Demonstrations
- Group Discussions
Training Agenda: Day 1

- Overview of the ST20 Toolset
- Installing the ST20 Toolset
- Working with projects
- Building with ST Visual Develop
- Debugging with ST Visual Develop
- ROM Debugging
- Trace & Profile
OS20 RTOS

- OS20 Kernel
- Tasks
- Memory Management
- Semaphores, Mutexes and Message Queues
- Interrupts
Overview of ST20 Toolset
Outline

- ST20 Micro Processor Core
- Diagnostic Control Unit (DCU)
- ST20 Micro Connect
- ST20 Toolset
- OS20 RTOS
- Development Environment

ST20 Embedded Software Toolset Training:
Objectives

At the end of this session the learner will be able to:

- List the key features of the ST20 Toolset.
- Describe the ST20 development environment.
ST20: 32-bit embedded micro core

Core = CPU + memory + interrupts + debug …

ST20-C2
- Base ST20 instruction set
- 174 instructions
- 2 priority h/w scheduler

ST20-C1
- Optimised instruction set
- 80 instructions
- DSP extensions

Example Core

ST BUS

INTS

CPU

DCU

ARBITER

I-cache

SRAM

D-cache
Full feature 32-bit embedded micro core
- Full ST20 architecture
- Hardware scheduling micro-kernel
174 instructions

ST20-C2 cores:
- ST20-C2 TPMAC 1st Generation Core
  - 2K $’s, 4K SRAM, DCU2, 100MHz, 4.9mm² (H8)
- ST20-C2xx 2nd Generation Cores
  - ST20-C200 - 4K $’s, 8K SRAM, DCU3, 120MHz, 5.29mm² (H8)
  - ST20-C201 - 8K $’s, 8K SRAM, DCU3, 200MHz, 3.57mm² (H9)
  - ST20-C202 - 8K $’s, 4K SRAM, DCU3, 200MHz, 3.60mm² (H9)
ST20-C201 Improved Caches

- ST20-C201 increased cache size from 4KB to 8KB
- Cache miss rate reduced by >40%
  - Higher CPU performance
  - Less memory traffic on STBUS

---

**EEMBC Benchmarks**

### I-Cache
- Miss rate
- 4KB: 15%
- 8KB: 4%

**47%**

### D-Cache
- Miss rate
- 4KB: 20%
- 8KB: 12%

**41%**

ST20 Embedded Software Toolset Training:
Specifications

- 2 way set associative Caches  
  New feature
- BISTs from ST CR&D generator  
  New feature
- Target CPU frequency: 200MHz
- Layout Area : 3.60 mm² (H9)
- Timer and interrupt handling
- Standard STBUS interfaces
- SRAM scaled down to 4K vs C201
ST20-C1 Family

- Smaller than ST20-C2
  - ST20-C1 CPU is 4x smaller than C2 CPU in layout area
- Optimised Instruction Set
- 80 Instructions
- DSP extensions
  - MAC (DSP)

<table>
<thead>
<tr>
<th>ST20-C1 Cores</th>
<th>I-cache (KB)</th>
<th>D-cache (KB)</th>
<th>SRAM (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C100</td>
<td>4</td>
<td>-</td>
<td>8</td>
</tr>
<tr>
<td>C101</td>
<td>32</td>
<td>32</td>
<td>128</td>
</tr>
<tr>
<td>C102</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>C103</td>
<td>2</td>
<td>2</td>
<td>64</td>
</tr>
<tr>
<td>C104</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>C105</td>
<td>8</td>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>

Fast RAM access
1 cycle write
1/2 cycle read
Specifications

- Single Cycle Cache Access  New feature
- BISTs from CR&D generator  New feature
- CPU frequency: 190 MHz
- Layout Area: 2.2mm² (H9)
- ST20-C1 CPU: 41 Kgates - Extremely Small
- Timer and interrupt handling
- Standard STBUS interfaces
- I-stream, D-stream and Peripherals
- DCU3: enhanced software debug features
- DCU3: TAPlink interface to host (via ST20 Micro Connect)
ST20 Core Roadmap

ST20-C2 Family
- ST20-C20x 200MHz
- ST20-C20x 190MHz
- ST20-C10x 190MHz
- ST20-C10x 225MHz
- ST20-C10x 200MHz
- ST20-C10x 108MHz
- ST20-TPMAC 100-166MHz

ST20-C1 Family
- ST20-C10x 225MHz
- ST20-C10x 200MHz
- ST20-C10x 190MHz
- ST20-C10x 108MHz

Technology:
- H8
- H9
- CMOS90

ST20 Embedded Software Toolset Training:
DCU – Diagnostic Control Unit

- On-chip silicon module adjacent to CPU
- Interface between the debugger and the CPU
- Access to memory independent of the CPU

Supports:

- Code download; reset and boot of target CPU
- Setting code and data breakpoints
- Program control
- Viewing program state
- Trace & Profiling
**Introduction of DCU3**

- **DCU3**
  - New features
  - Introduced in the ST20-C200
  - Improved performance

**ST20 Embedded Software Toolset Training:**
DCU3 vs DCU2

- Improved performance and flexibility:
  - High speed clock rate (166MHz+)
  - ST20 Micro Connect bandwidth improvements
  - General comparators for code/data breakpoints

- Improved functionality:
  - Easier trace buffer placement.
  - Time-stamping of trace records.
    - Using cycle count.
  - Capture information on events
    - e.g. precise Ipdr on watch point.
  - Can collect trace of Wptr, Address or Data.
  - Compression of trace data.
    - Based on changes since last record
  - Limit events/trace collection to specific task workspace

ST20 Embedded Software Toolset Training:
ST20 Toolset

- ST Visual Debug IDE
  - Project workspace management
  - Program Release or Debug mode
  - Integrated Editor from *Tetradyne*
- Embedded programming
  - Full ANSI C/C++
  - Support for multiple ROMs
- Advanced Debugging
  - Exploits DCU on-chip emulation features
  - Real time trace capture
  - Sampling program profiler
- OS20 Real time kernel
ST20 Tools Roadmap

- **Release R1.9**
  - Support for DCU3 devices
  - st20run GUI Debugger
  - First Linux release

- **Release R2.0**
  - New **ST Visual Develop** IDE
  - Full ISO C++ including STL
  - New and enhanced tracing features

ST20 Embedded Software Toolset Training:
Key Features

- Multi-priority pre-emptive scheduler.
  - 16 Levels of priority
  - Time slicing of tasks within same priority level.
- Semaphores, Mutexes, Message Queues and Timers.
- Interrupt and Memory Management.

Light weight (~10KBytes) and efficient with fast context switching.

Bundled with toolset – source supplied Royalty Free.
Third Party Software

- NexGen (www.nexgen-software.com)
  - TCP/IP stack
- ANT (www.ant.co.uk)
  - FRESCO browser
- Teamlog (www.teamlog.fr)
  - Common Interface stack
- SmartLink (www.smlink.com) / Gao Research (www.gaoresearch.com)
  - V.22bis software modem
- Agere (www.agere.com)
  - Controllerless modem (speeds up to V.92)
- SoftConnex (www.softconnex.com)
  - External chip today, Integrated with core next year
  - USB1.1 driver, USB 2.0 coming soon
Development Environment

ST20 Embedded Software Toolset Training:

Host

ST Micro Connect

Target

JTAG Interface

ST20
In this session we have overviewed the ST20, DCU and ST20 Toolset.

We have also looked at the development environment and the OS20 multi-tasking Operating System.

… next we learn how to Install the ST20 Toolset
Installing the ST20 Toolset
Installing the ST20 Toolset

R2.0 Supported Platforms
- Windows 98/2000/NT4/XP.
- Red Hat Linux V8 (or later).
- Sun4 Solaris platforms running Solaris 2.6 (or later).

The PC Windows, PC Linux and Sun Solaris release is supplied on a CD-ROM, or from the STMicroelectronics FTP site.

The “Delivery Manual”, provided with each un-installed release package, describes in detail how to install the ST20 Toolset.
Exercise 1

- In this exercise you will install the ST20 Toolset on your PC/Work Station.
- See handout for detailed instructions
Working with projects
Introduction to workspaces and projects
Creating a workspace
Adding a project
Adding files to a project
Setting Project Dependencies
Exercise
At the end of the session the learner will be able to:

- Create a workspace
- Add projects to a workspace
- Add files to a project
- Set project dependencies
Introduction to workspaces and projects

- Application development and debugging is organised into workspaces.
- A workspace can contain a number of projects.
- Each project can be dependent on other projects thereby creating a hierarchal structure.
Introduction to workspaces and projects

- Workspaces contain the information required to debug an application.
- Projects contain the information necessary to build an application.
- There are typically two information sets:
  - Project settings.
    - Consists of the information necessary for a successful build of an application.
  - Debugging settings.
    - Includes information such as the target, breakpoints, memory mapping and trace.
Creating a workspace

There are four methods that can be used to create a workspace:

1. Using the **New Workspace Wizard**.
2. Creating the workspace from an existing project.
3. Creating the workspace from an executable file.
4. Creating the workspace from a makefile.
Enter a name to use for the Workspace filename.

Enter or browse for the Workspace location.
There are two methods that can be used to add a project to the workspace:

- Create a new project and add it to the workspace.
- Add an existing project to the workspace.
There are three methods that can be used to create a project:

1. Using the **New Project** window.
2. Creating the project from an executable file.
3. Creating the project from a makefile.
- Enter a name for the Project filename.
- Enter or browse for the Project location.
Using the Text Editor

- Click on the New Text File icon:

![Text Editor Interface]

ST20 Embedded Software Toolset Training:
Do one of the following:

1. Click on the project and select Insert Files into Project... from the Project menu.
2. Right click on the project and select Add Files to Project...
3. Right click on the project folder (Source Files or Include Files) and select Add Files to Folder...
Multiple projects with the same workspace

```c
#include "main.h"

extern int func1(void);

int main (void)
{
    func1();
}
```
Setting the Active Project

- In a workspace that contains more than one project, there is only one currently active project.
- The active project is the project that, for example, is built when **Build** is selected from the Build toolbar.
- When an active project is built, any dependencies of the active project are also built.
- In ST Visual Develop, the active project is displayed in **bold**.

ST20 Embedded Software Toolset Training:
Setting project dependencies

- Dependencies can exist between projects within the same workspace.
- If a project is dependent on another project, it is not built until the project that it is dependent on is built.
- Dependencies are created and modified using the **Project Dependencies** window.
Exercise 2

In this exercise you will...

- Create a New workspace and project.
- Write a ‘simple’ C program using the IDE text editor.
- Add the C program to the project.
- Save and close the workspace.
- Create a second workspace that contains multiple projects.

See handout for detailed instructions
In this session you have learned how to …

- Create a workspace
- Add projects to a workspace
- Add files to a project
- Set project dependencies

… next we will learn how to build an application using ST20 Visual Develop
Building with ST Visual Develop
Outline

- Introduction to Building
- st20cc – compile & link
- Build configuration settings
- Setting the options for building
- Building the application
- Configuration files (part 1)
- Building a library
- Exercise
At the end of the session the learner will be able to:

- Configure the build settings
- Set the options for building
- Build an application against a set of configuration files
- Build a library
An application is built to generate either…

- A ROM image suitable for blowing into ROM.
- A development system in the form of a linked unit (LKU).
- A Relocatable Code Unit (RCU) that can be dynamically loaded at run-time.

To build an application, ST Visual Develop calls the compile/link tool st20cc.

ST Visual Develop is used to…

- Chose between the different build configuration settings.
- Set the build options that are passed to st20cc.
To build an application, ST Visual Develop calls the compile/link tool st20cc.
Embedded Programming

- Compiler
  - Full ANSI C/C++ language implementation.
  - Advanced compiler optimisations.
  - Support for interrupts, traps, assembler inserts.

- Linker
  - Fine grained code and data placement.
  - Relocation of code from ROM to RAM.
  - Support for multiple ROMs.
  - EMI configuration.
  - Dynamic loading of relocatable code.
  - Controlled by commands from “.cfg” configuration file.
Support for C++

- Full ISO C++ including STL
  - Iostreams & Standard Template Library
    - Provided by *Dinkumware*
  - Two variants of C++ runtime system
    - Full ISO (with/without exceptions)
    - Embedded (with/without exceptions)

When processing C++ code, st20cc invokes the Edison Design Group (EDG) preprocessor to convert C++ into C.
There are two configuration settings that are provided by default:

- Debug settings (output files are created in a **Debug** subdirectory)
- Release settings (output files are created in a **Release** subdirectory)
Before you can build an application you must specify the build options.

These options control how st20cc builds the application.

The build options are set in the **Project Settings** window.

To open the **Project Settings** window:

Select the project or set the project to be the active project.
C/C++ Compiler

- Category
  - General
  - Files
  - Code Generation
  - Errors & Warnings
  - Runtime checks
  - Customise
Linker

- Category
  - General
  - Files
  - Libraries & Objects
  - Map files
  - Warnings
  - ROM
  - Dynamic Library & RCU
  - Customise

Project Settings Window

[Diagram of Linker settings window with various options and settings]

ST20 Embedded Software Toolset Training:
Building the application

There are several ways in which the application can be built:

- Compiling an individual source file.
- Building a project.
- Rebuilding all projects.

Each of these options compile/build the active project and any projects on which the active project is dependent.

The results of a build are displayed in the Build tab of the Output window.
Compiling an individual file

Select the source file to build or open it in a separate Editor window.
Note: If dependencies exist for the selected project then the projects on which it is dependent are built first even if they have not been selected.
Rebuilding and cleaning projects

- Click on the following icon to rebuild the active project without making any changes or removing the output files...

- To remove the intermediate build files from a project, perform...

ST20 Embedded Software Toolset Training:
Configuration files (part 1)

- Configuration files use commands that are used to describe:
  - Target memory **segments**
  - Placement of code and data **sections**

- Segments and Sections
  - Target hardware is described in terms of memory **segments**.
  - A **segment** is a contiguous region of memory.
  - An application is described in terms of **sections** generated by the compiler.
  - A **section** is a contiguous block of code or data.
  - A configuration file is used to specify which code and data **sections** to place into which **segments**.

ST20 Embedded Software Toolset Training:
Example configuration file

```
mb361.cfg

proc MemSpace {
    chip STi5516
    memory EXTERNAL 0x40000000 (16 * M) RAM
    memory FPGA 0x70000000 (16 * M) DEVICE
    memory FLASH 0x7FC00000 (4 * K) ROM
    memory SDRAM 0xC0000000 (8 * M) RAM
    PlaceDebugTrapHandler EXTERNAL
}

proc link5516 {
    MemSpace
    stack EXTERNAL
    heap EXTERNAL
}
```

Main build procedure
The **chip** command is used to declare core and chip specific information to the toolset.

```
e.g.   chip STi5516
```

The core specific information includes:
- the processor type (C1/ C2).
- details of the on-chip memory.
- the diagnostic controller unit (DCU2/DCU3).
- the name of CPU registers.
- Cache and interrupt management.

Supported chips include:
- STi5514, STi5516, STi5517, ST205518, STi5519, STi5528, STi5580, STi5588, STi5589, STi5598, STV0396, STV3500, ST20GP6, ST20DC2, ST20C1SIM, ST20C2SIM among others...
Stack and Heap Commands

- Stack is specified using the `stack` command:
  
  \[
  \text{SegmentName} \quad \text{StackSize} \\
  \text{stack} \quad \text{INTERNAL} \quad 4096
  \]
  
  - **MUST** be specified in the command file used for linking.
  - If the size is not specified, the stack takes up the remainder of the specified segment after code/data placement is complete.

- Heap is specified using the `heap` command:
  
  \[
  \text{SegmentName} \quad \text{StackSize} \\
  \text{heap} \quad \text{INTERNAL} \quad 4096
  \]
  
  - Necessary only if malloc and free are used in your program.

- Special case:
  
  \[
  \text{stack} \quad \text{INTERNAL} \\
  \text{heap} \quad \text{INTERNAL}
  \]
  
  - Stack and heap equally share what is left of the INTERNAL segment after code/data placement is complete.
The debug traphandler is downloaded by the debugger at runtime and is used to handle breakpoints.

At the link stage we must reserve space in memory for the debug traphandler.

Can use either the `PlaceDebugTrapHandler` command:

```
PlaceDebugTrapHandler EXTERNAL
```

Alternatively a debug memory segment can be defined using the `memory` command:

```
memory TRAPHANDLER 0x80002000 1024 DEBUG
```

Notes: Must be named ‘TRAPHANDLER’.

Debug segment should be 1024 bytes long.
In the absence of any other instructions the linker performs default placement.

All code, data, bss and const sections are combined together into their contiguous default sections:

<table>
<thead>
<tr>
<th>def_code</th>
<th>All code</th>
</tr>
</thead>
<tbody>
<tr>
<td>def_data</td>
<td>All initialised non-constant data</td>
</tr>
<tr>
<td>def_bss</td>
<td>All non-initialised non-constant data</td>
</tr>
<tr>
<td>def_const</td>
<td>All constant data</td>
</tr>
</tbody>
</table>

Any unplaced sections, including the default sections, are placed into the largest defined RAM segment.
Placement of sections is achieved using the `place` command

e.g.
```c
place <app.txt> EXTERNAL
place def_data INTERNAL
```

Named sections generated by use of the `#pragma ST_section`, can also be placed using the `place` command.

```c
int fast_fn(int x);

#pragma ST_section(fast_fn, "fast_code")
int fast_fn (int x)
{ return x + 2; }
```

```c
proc myplace {

#pragma ST_section(fast_code, "fast_code")
place def_data INTERNAL
place def_bss INTERNAL
place def_code DRAM
place def_const DRAM
stack INTERNAL
heap DRAM
}
```
Including configuration files

1. Set the main configuration file and the directories in which to search for all included configuration files:

2. Set the **build procedure**: 

   ![Build Procedure Screenshot]

   - **Output File Name**: $(ProjectSFile).lk
   - **Build Procedure**: link5516
Generating a MAP file

- This can be done through the **Project Settings Window**.
# Example MAP File

## Section/Region map

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Size</th>
<th>Type</th>
<th>Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>fast_code</td>
<td>0x80000140</td>
<td>8</td>
<td>CODE</td>
<td>INTERNAL</td>
</tr>
<tr>
<td>def_bss</td>
<td>0x80000148</td>
<td>320</td>
<td>BSS</td>
<td>INTERNAL</td>
</tr>
<tr>
<td>def_data</td>
<td>0x80000288</td>
<td>172</td>
<td>INIT</td>
<td>INTERNAL</td>
</tr>
<tr>
<td>stack</td>
<td>0x80000334</td>
<td>7564</td>
<td>STACK</td>
<td>INTERNAL</td>
</tr>
<tr>
<td>def_code</td>
<td>0x40000000</td>
<td>3904</td>
<td>CODE</td>
<td>DRAM</td>
</tr>
<tr>
<td>def_const</td>
<td>0x40000f40</td>
<td>4</td>
<td>CONST</td>
<td>DRAM</td>
</tr>
<tr>
<td>bootdata</td>
<td>0x40000f44</td>
<td>24</td>
<td>BOOTDATA</td>
<td>DRAM</td>
</tr>
<tr>
<td>heap</td>
<td>0x40000f5c</td>
<td>258212</td>
<td>HEAP</td>
<td>DRAM</td>
</tr>
</tbody>
</table>

## Symbol map

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Type</th>
<th>Section</th>
<th>Source</th>
<th>Mod/Lib</th>
</tr>
</thead>
<tbody>
<tr>
<td>fast_fn</td>
<td>0x80000140</td>
<td>G</td>
<td>fast_code</td>
<td>test.c</td>
<td>test.tco</td>
</tr>
<tr>
<td>bill</td>
<td>0x80000148</td>
<td>G</td>
<td>def_bss</td>
<td>test.c</td>
<td>test.tco</td>
</tr>
<tr>
<td>main</td>
<td>0x40000000</td>
<td>G</td>
<td>def_code</td>
<td>test.c</td>
<td>test.tco</td>
</tr>
<tr>
<td>jack</td>
<td>0x40000f40</td>
<td>G</td>
<td>def_const</td>
<td>test.c</td>
<td>test.tco</td>
</tr>
</tbody>
</table>
Building a Library

- A separate project is added to a workspace for each library that is to be managed within st20dev.
- The **project setting window** is used to specify that a project is to output a static library.
Including libraries and/or Objects

Again, this is done through the Project Settings Window.
Exercise 3

In this exercise you will...

- Set the Project Settings to build an application for the STi5516 Eval board.
- Using the configuration files provided you will practice...
  - Placing sections of code in memory segments.
- Generate a Map file.
- Build a library

See handout for detailed instructions
In this session you have learned how to…

- Configure the build settings
- Set the options for building
- Build an application against a set of configuration files
- Build a library

… next we will learn how to debug using ST20 Visual Develop
Debugging with
ST Visual Develop
Outline

- Introduction to debugging
- Setting the options for debugging
- Configuration files (part 2)
- Hardware Initialisation
- Starting Debugging
- Main features of the ST20dev debugger
- Exercise
At the end of the session the learner will be able to:

- Set the options for debugging
- Use Configuration files to perform hardware initialisation
- Start the Debugger
ST Visual Develop can be used to debug...
- A boot from ROM application
- A development application in the form of a linked unit (LKU).
- A Relocatable Code Unit (RCU).

The ST Visual Develop will interface to a Hardware Target using the ST20 Micro Connect via...
- Ethernet
- Parallel Port (PCs)
- USB (PCs)

The ST Visual Develop can also run programs on the ST20 Instruction set simulator (st20sim).
Setting the options for debugging

- Once you have built an application, you must specify the debug options.
- These options control connecting to the target and how to debug the application.
- The debug options are set in the **Workspace Settings** window.
- To open the **Workspace Settings** window click on the icon.
Enter the **Target** to connect to, as defined in the configuration file.

Use the **Include files** area to enter or browse for any configuration files.

Use the **Source directories** area to enter or browse for the directories that contain any configuration files specified in the **Include files** option.
The **target** command is used in a configuration file to connect to a real or simulated target.

---

<table>
<thead>
<tr>
<th>Target</th>
<th>Target Command Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ethernet</td>
<td><code>target mytarget tap &quot;jei_soc ip_addr&quot;</code> connect proc</td>
</tr>
<tr>
<td>Parallel port</td>
<td><code>target mytarget tap &quot;hti_ppi lpt1&quot;</code> connect proc</td>
</tr>
<tr>
<td>USB</td>
<td><code>target mytarget tap &quot;hti_usb usb&quot;</code> connect proc</td>
</tr>
<tr>
<td>Simulator</td>
<td><code>target mysim st20sim &quot;st20sim -q -f st20hw.cfg -p c2def&quot;</code> config_proc</td>
</tr>
</tbody>
</table>

Note: **TAP** - Test Access Port
- Hardware requires initialisation before an application can be run.
- This is achieved by a procedure that is called at the end of a `target` command line.

```app.cfg
## Routine called from TARGET definition
proc board_runtime_init {
    reset
    MemSpace
    ST20C2MemoryInit
    EMIpokes
    ConfigureSDRAM
}

## TARGETS
target mytarget tap "jei_soc ip_addr tckdiv=4" board_runtime_init
```
Hardware Initialisation is required for a DCU booted application (LKU).

Hardware Initialisation is included as part of the bootstrap on ROM booted application.

Hardware Initialisation:
- External Memory Interface (EMI).
- Shared Memory Interface (SMI).
- Clearing critical areas of memory (CPU Registers).

The `reset` command is used to reset and stall the CPU before an application is loaded.

Finally, a procedure is called that describes to the debugger the connected device and the address space.
Hardware Initialisation of EMI

- A call is made to a procedure containing a ‘poke table’ which makes use of the `poke` command:
  
  ```
  poke Register_Address Register_Value
  ```

- This will set up the EMI register values to allow access to external memory.

```
app.cfg

...  
## Routine called from TARGET definition  
proc board_runtime_init {  
    reset  
    MemSpace  
    ST20C2MemoryInit  
    EMIpokes  
    ConfigureSDRAM  
}  
...
```

```
proc EMIpokes {  
    ##BANK 0 DRAM  
    poke -d 0x00002000 0x132a  
    poke -d 0x00002004 0xffff8  
    poke -d 0x00002008 0x012a  
    poke -d 0x0000200C 0x095a  
    ##BANK 1 SDRAM  
    poke -d 0x00002010 0x132a  
    ...  
}
```
SMI Initialisation is slightly more complex than EMI initialisation.

Like EMI initialisation, the SMI registers are programmed through a series of pokes to allow access to shared SDRAM.

However, before this can be done, an iterative series of peek and pokes are used to test the phase of the SDRAM PLL (Phase Lock Loop) and lock to that phase as soon as a test returns true.

An example of a configuration procedures used for SMI SDRAM initialisation is found within the practicals directory…

..\ST20Training(R2.0)\practicals(5516)\5516cfg\sti5516sd.cfg
A call is made to a procedure that will clear critical areas of memory (i.e. CPU Registers).

```app.cfg

... proc board_runtime_init { reset MemSpace ST20C2MemoryInit EMIpokes ConfigureSDRAM } ...

proc ST20C2MemoryInit mint = 0x80000000 trapstart = mint+0x40 {
  ## clear all words below trapstart to MOSTNEG INT for (i = mint; i < trapstart; i = i + 4) {
    poke (i) (mint)
  }
}
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 `target` command provides the ability to set the clock speed of the JTAG interface.

This is set by adding an assignment to the `tckdiv` parameter to divide the JTAG clock frequency.

This parameter takes powers of 2 and the default value is 1.

E.g.

```
target mytarget tap "jei_soc ip_addr tckdiv=4" config_proc
```
Starting Debugging

- After building an application and setting the debug options it can be debugged.
- To start debugging click on the icon.

![Debugging Icon](image)

LED 0520 App - ST Visual Develop task "root" [stopped by breakpoint at cos20_le.c 16:0; target stopped...]

`/* task data */
int main(void)

  task_t *task_list[2];
  device_id_t devid = device_id();
  printf("--- 0820 LED Practical ---\n
task 0 "root" stopped at cos20_le.c 16:0 by breakpoint 0

168463756
1450767003
905419696

SDRAM locked on phase 2
SDRAM init has been successful
Remote execution of command: runtime o2os20
  task 0 "root" stopped at cos20_le.c 16:0 by breakpoint 0

Content: Program 0, Task 0

ST20 Embedded Software Toolset Training:
Main Features of the Debugger

- Open multiple Source Windows
- View and Dock various windows:
  - View Workspace
  - Disassembly Window
  - Symbol Browser
  - Task Window
  - Memory Window
  - Registers Window
  - Callstack
  - Watch Window
  - Events Window
  - Map Window
- Code and Data breakpoints
- Real time trace capture
- Sampling program profiler
- ROM program debugging

ST20 Embedded Software Toolset Training:
Source Window

- Open multiple source windows.
- View Execution pointer.
- View Code and data breakpoints.
- Variable tooltips.
- Right click for Debug Menu.

ST20 Embedded Software Toolset Training:
Disassembly Window

- Opened by clicking on the icon.
  - Step in disassembly view.

```
Disassembly

0x4000007a ldnl 0
0x4000007b ldnl 4
0x4000007c cj 0x40000096  ## <stack.c 57 0> in recSTACK
0x4000007e ldl 3
0x4000007f ldnl 0
0x40000080 stl 0
stack.c 52 *i = (*s)->elem ;

0x40000081 ldl 3
0x40000082 ldnl 0
0x40000083 ldl 4
0x40000084 ldc 16
0x40000086 move
stack.c 53 *s = (*s)->next ;
0x40000088 ldl 3
0x40000089 ldnl 0
0x4000008a ldnl 4
0x4000008b ldl 3
0x4000008c stnl 0
stack.c 54 free(t) ;
```

ST20 Embedded Software Toolset Training:
Symbols Window

- Opened by clicking on the icon.
  - Open Source (where applicable)
  - Insert Breakpoint
Map Window

- Opened by clicking on the icon.
  - Open Source (where applicable)
  - Insert Breakpoint
Opened by clicking on the icon.

**Events Window**

- **Add**
- **Modify**
- **Remove**
- **Open Source**
- **Docking View**
- **Hide**

**Edit Event 4**

- **Event type:** Data Value
- **Enabled**
- **Size:** Data Value
- **Value:** 15

**Context:**
- **Program:** D:\play3\Debug\os20_led.lku
- **Task:** Any task
- **Frame:**

- **Advanced...**
- **OK**
- **Cancel**

ST20 Embedded Software Toolset Training:
There are two types of Breakpoints; Software and Hardware.

- **Software Breakpoints:**
  - Code breakpoint normally implemented with a special ‘j 0’ instruction (for RAM):

- **Hardware Breakpoints:**
  - The DCU has a set of registers for:
    - Code breakpoints (normally for ROM)
    - Ranged breakpoints.
    - Data breakpoints.

Hitting a breakpoint causes the debug trap handler to take control.

Debug trap handler is downloaded at runtime.

Interrupts are disabled whilst in trap handler.

Trap handler signals the host.
Opened by clicking on the icon.

- Drag and drop variables into Watch Window
Opened by clicking on the icon.
- Add / Remove / Format Registers
Opened by clicking on the icon.
- Open multiple memory windows
- Format: Hexadecimal \ Binary \ Decimal \ ASCII …
- Type: Instructions \ Strings \ Statements \ Char \ Short …

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4080266c</td>
<td>0x40ffdb68</td>
</tr>
<tr>
<td>0x40802670</td>
<td>0x40ffd730</td>
</tr>
<tr>
<td>0x40802674</td>
<td>0x40ff2f68</td>
</tr>
<tr>
<td>0x40802678</td>
<td>0x40ffcec0</td>
</tr>
<tr>
<td>0x4080267c</td>
<td>0x40ffa88</td>
</tr>
<tr>
<td>0x40802680</td>
<td>0x40ff650</td>
</tr>
<tr>
<td>0x40802684</td>
<td>0x40001e06 &lt;cstart2.lib&gt; in cstart2()</td>
</tr>
<tr>
<td>0x40802688</td>
<td>0x4000405c</td>
</tr>
<tr>
<td>0x4080268c</td>
<td>0x00000001</td>
</tr>
</tbody>
</table>
```
Task Window

- Opened by clicking on the icon.

![Task Window Image](image)

```
Tasks
0 "root" stopped by breakpoint at <os20.c 47 0>
1 "Trap[Low][Scheduler]" inactive last stopped at 0x8000140
2 "Letter 'A'" terminated last stopped at 0x8000000
3 "Letter 'B'" active last stopped at <return>S 43 0>
4 "Letter 'C'" active last stopped at <return>S 43 0>
5 "Letter 'D'" active last stopped at <return>S 43 0>
6 "Letter 'E'" active last stopped at <return>S 43 0>
7 "Letter 'F'" terminated last stopped at 0x8000000

Task details
0 "root" stopped by breakpoint at <os20.c 47 0> in space 0
program 0 D:\\play3\Debug\Letters.lik
stack 0x4000520c - 0x40802708
OS/20: task = 0x4000471c, tdesc = 0x80000798
OS/20: priority = 15
iptr = 0x40000214
wptr = 0x40802650
areg = 0x00000018
breg = 0x0000f00
creg = 0x00000000
```
Opened by clicking on the icon.
Command Console Window

- Opened by clicking on the icon.
  - Enter commands directly or include a command file.
Quick method to debug and/or run an existing LKU

To quickly use the st20dev debugger to debug an existing LKU application, invoke the command line...

```
st20dev existing.lku -t mytarget -i app.cfg -g
```

To quickly run an existing LKU application, invoke the command line...

```
st20run existing.lku -t mytarget -i app.cfg
```

*NB: the st20run -g option is no longer supported*
Exercise 4

In this exercise you will…

- Debug a program through the st20dev debugger.
- Explore the debugger capabilities.

See handout for detailed instructions
In this session you have learned how to...

- Set the options for debugging
- Use Configuration files to perform hardware initialisation
- Start the Debugger

... Next we will learn how to debug a boot from ROM application.
ROM Debugging
Outline

- Hardware Initialisation
- Building a ROM image
- Boot from ROM Sequence
- Controlling ROM images
- Flash programming
- ROM Debugging
- Exercise
At the end of the lesson the learner will be able to:

- Build a ROM image file
- Burn the ROM image to FLASH
- Debug a boot from ROM application
Hardware Initialisation

- Hardware Initialisation is required for a DCU booted application (.lk). The same is required when booting an application from ROM.
- The bootstrap on ROM booted application is called upon to initialise:
  - EMI/SMI
  - Clear critical areas of memory
- When building to generate a ROM image, the command procedures that are used to initialise both the EMI registers and the clearing of critical areas of memory must be called.
- Both procedures are made up of a series of pokes and these are parsed into the bootdata section of the bootstrap automatically.
- On the other hand, since configuration file SMI initialisation involves conditional peek and pokes, it can not be automatically added to the bootstrap by the linker.
- Instead ‘C’ code is written to perform SMI Initialisation and this is supplied to a tools bootstrap stub function.
When linking to generate a `.lku`

st20cc -p link_lku

When linking to generate a ROM image

st20cc -p link_rom

---

**app.cfg**

```plaintext
... 
proc link_lku {
  hw
    stack EXTERNAL
    heap EXTERNAL
}

proc link_rom {
  MemSpace
  EMIpokes
  ST20C2MemoryInit
    stack EXTERNAL
    heap EXTERNAL
}

proc MemSpace {
  chip STi5516
  memory EXTERNAL 0x40000000 (16* M) RAM
  ...
}

proc EMIpokes {
  poke -d 0x00002000 0x132a...
  ...
}

proc ST20C2MemoryInit
  mint = 0x80000000
  trapstart = mint+0x40 {
    for (i = mint; i < trapstart; i = i + 4) {
      poke (i) (mint)
    }
  }
}
```
Two stub functions `PrePookLoopCallback()` and `PostPokeLoopCallback()` are provided in a source file `initfuncs.c`, a copy of which can be found under...

```plaintext
...\ST20R2.0.5\src\cstartup\initfuncs.c
```

Either function can be used to supply ‘C’ code to perform SMI initialisation.

Add `initfuncs.c` to the workspace so that it is included in the build.
Open the Project Settings window:

- Enter the Output File Name (if required)
- Enter the main build procedure
- Select the Output File Type (hex, binary or Motorola)
Example of 512K of FLASH on the C2 core:

Iptr: 0x7fffffff  
(RESET)

FLASH

0x7fffffff

startblock

bootdata

def const

<shutdown_section>

<startup_section>

def_data <partial>

<initfuncs_text>

<romload2_text>

<romload_text>

def_code

0x7ff80000
- Resetting the chip causes it to jump to its Boot vector:
  - C2 – fixed at 0x7fffffff
  - C1 – chip dependent, use bootiptr command to specify.

- Bootstrap code
  - Sets up a temporary workspace for itself in internal memory (the size of which can be controlled by using the initstack command)
  - Calls PrePokeLoopCallback function of initfuncs.c
  - Walks “poke” table performing each poke
  - Calls PostPokeLoopCallback function of initfuncs.c
  - Sets-up application Wptr to point to location specified with stack command
  - Walks section table
  - Moves (copies) code and data sections from ROM to RAM
  - Initialises DATA segments and zeros BSS sections
  - Jumps to application start-up code
### Example ROM Map file

#### Section/Region map

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Size</th>
<th>Type</th>
<th>Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>def_code</td>
<td>0x40000000</td>
<td>3836</td>
<td>CODE</td>
<td>DRAM</td>
</tr>
<tr>
<td>def_bss</td>
<td>0x40000efc</td>
<td>320</td>
<td>BSS</td>
<td>DRAM</td>
</tr>
<tr>
<td>def_data</td>
<td>0x4000103c</td>
<td>172</td>
<td>INIT</td>
<td>DRAM</td>
</tr>
<tr>
<td>def_const</td>
<td>0x400010e8</td>
<td>4</td>
<td>CONST</td>
<td>DRAM</td>
</tr>
<tr>
<td>stack</td>
<td>0x400010ec</td>
<td>128908</td>
<td>STACK</td>
<td>DRAM</td>
</tr>
<tr>
<td>heap</td>
<td>0x40020878</td>
<td>128904</td>
<td>HEAP</td>
<td>DRAM</td>
</tr>
<tr>
<td>startblock</td>
<td>0x70000000</td>
<td>24</td>
<td>START</td>
<td>FLASH</td>
</tr>
<tr>
<td>&lt;romload_text&gt;</td>
<td>0x70000018</td>
<td>68</td>
<td>CODE</td>
<td>FLASH</td>
</tr>
<tr>
<td>&lt;romload2_text&gt;</td>
<td>0x7000005c</td>
<td>136</td>
<td>CODE</td>
<td>FLASH</td>
</tr>
<tr>
<td>def_code</td>
<td>0x700000e4</td>
<td>3836</td>
<td>MVTORAM</td>
<td>FLASH</td>
</tr>
<tr>
<td>def_data &lt;partial&gt;</td>
<td>0x70000fe0</td>
<td>4</td>
<td>MVTORAM</td>
<td>FLASH</td>
</tr>
<tr>
<td>def_data &lt;partial&gt;</td>
<td>0x70000fe4</td>
<td>16</td>
<td>MVTORAM</td>
<td>FLASH</td>
</tr>
<tr>
<td>def_data &lt;partial&gt;</td>
<td>0x70000ff4</td>
<td>4</td>
<td>MVTORAM</td>
<td>FLASH</td>
</tr>
<tr>
<td>def_const</td>
<td>0x70000ff8</td>
<td>4</td>
<td>MVTORAM</td>
<td>FLASH</td>
</tr>
<tr>
<td>section-table</td>
<td>0x70000ffc</td>
<td>96</td>
<td>SECTTAB</td>
<td>FLASH</td>
</tr>
<tr>
<td>bootdata</td>
<td>0x700105c</td>
<td>208</td>
<td>BOOTDATA</td>
<td>FLASH</td>
</tr>
</tbody>
</table>

ST20 Embedded Software Toolset Training:
Default addresses in ROM

- Un-placed sections are stored into ROM where they will fit and copied to RAM by the loader.
- Bootdata and section-table information is always in ROM
- ROM loader code is placed in ROM
- Address in ROM image are absolute addresses in target address space
- Unused space is left empty
In a ROM based system each section has (potentially) two locations:

- Location in ROM.
- Location at run-time.

**place** commands controls the location at run-time.

- Code placed in ROM will run from ROM.
- Constant data placed in ROM will not be copied to RAM.

**store** command controls location in ROM.

- Allows the user to control which ROM contains a specific section in systems with multiple ROM segments.

**bootdata** command controls which ROM contains the bootdata.
Example using ROM control

```
app.cfg

apphw
place fast_code  INTERNAL
place def_data   INTERNAL
place def_code   EPROM1
place def_const  EPROM2
place def_bss    DRAM
bootdata        EPROM2

store fast_code EPROM1
store def_data  EPROM2

stack INTERNAL
heap DRAM
```

ST20 Embedded Software Toolset Training:
- Can be done using a FLASH-burner program, which is run on the target as an LKU.
- The FALSH-burner program will then take the generated ROM Image found on the host and burn it FLASH.
- Easiest to use `st20run` to run the program:

```bash
st20run flashburner.lku -t mytarget -i app.cfg
```
- All you need is a connection from the on-chip DCU to your host.
- Two ways to debug a ROM system:
  1. Debug from reset (before the ROM bootstrap has begun to execute).
  2. Debug an application that has already booted and is running.

ST M
Micro Connect
Target
Trigger
---
---
Diagnostic
Control
Unit
User application program
Target Hardware ST20 based
Simply connect with a reset!

```bash
## Routine called from TARGET definition
proc board_runtime_init {
    reset
    MemSpace
    ST20C2MemoryInit
    EMIpokes
    ConfigureSDRAM
}

## TARGETS
target mytarget tap "jei_soc ip_addr tckdiv=4" board_runtime_init
```

Note: We load the dbg debug file

ST20 Embedded Software Toolset Training:
Debugging a booted application

- Simply connect without a reset!

```plaintext
mb361.cfg

## Routine called from TARGET definition
proc board_runtime_init_nr {
    MemSpace
    informs -enable
}

## TARGETS
target mytarget_nr tap "jei_soc ip_addr tckdiv=4" board_runtime_init_nr
```

Note: We load the dbg debug file

![Workspace Settings](image)
Exercise 5

- In this exercise you will …
  - Build the application as a ROM image file
  - Burn the ROM image to FLASH
  - Debug the boot from ROM application

- See handout for detailed instructions
In this Session you have learned how to...

- Build a ROM image file
- Burn the ROM image to FLASH
- Debug a boot from ROM application

... Next we will learn how to use the Trace and Profile debugging features.
Micro Core & Development Tools

Trace & Profile
Outline

- Introduction to Trace & Profile features
- Trace & Profile Window
- Trace Settings Window
- Trace Scenarios
- Post-mortem Debugging
- The trace buffer
- Exercise
At the end of the lesson the learner will be able to:

- Use the Trace & Profile features of the Tools.
- Be able to perform Post-mortem debugging.
- Customise the trace buffer.
The toolset supports the trace features of DCU.

The DCU can be configured to record key data, such as the IPTR value, when certain CPU events occur, such as a jump or a memory access.

This way, the history of the program execution may be examined sometime later.

The DCU writes the records to a trace buffer located in target memory.

This allows for all ST20 chips to support trace as no extra hardware is required.
If trace has been enabled and there is an application crash that requires a system reset, then the post-mortem feature can be used.

This allows the trace buffer to be examined to get some clues about what the CPU was doing before the crash.
The time spent in functions can be monitored using the profiler to sample the IPTR.

To complement this, trace on DCU3 can provide detailed and accurate timings, for example, the time spent in specific tasks or the time between two points can be examined.
The Trace and Profile Window is opened by clicking on the icon.
Trace Settings Window

- Acquire Settings
  - Scenario (Tasks, Jumps …)
  - Sensitivity (Context, Jumps …)
  - Records (To IPTR, From …)
  - Break event id

- Trace buffer location

- Action on buffer full
  - Wrap/Stop/Download

- Trace system intrusiveness
  - Stall CPU
  - CPU not stalled – may lose records

- Trace Acquisition Control
  - Start/stop trigger against event
### Trace Scenarios

<table>
<thead>
<tr>
<th>Trace Scenario</th>
<th>DCU type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tasks</td>
<td>DCU3</td>
<td>Task switches and function calls</td>
</tr>
<tr>
<td>Jumps</td>
<td>DCU3, DCU2</td>
<td>Jumps made by an application and identify function calls</td>
</tr>
<tr>
<td>All Code</td>
<td>DCU3</td>
<td>Record the address of all code fetches (IPTR) to analyse in detail a critical region of code</td>
</tr>
<tr>
<td>All Memory</td>
<td>DCU3</td>
<td>Record the address of all data memory accesses</td>
</tr>
<tr>
<td>Break</td>
<td>DCU3</td>
<td>Record when a breakpoint occurs</td>
</tr>
<tr>
<td>Break Range</td>
<td>DCU3</td>
<td>Monitor a region of memory and record the address of all accesses within the range</td>
</tr>
<tr>
<td>Break Value</td>
<td>DCU3</td>
<td>Monitor memory accesses to a symbol and record the value written to it</td>
</tr>
<tr>
<td>Custom Trace</td>
<td>DCU3, DCU2</td>
<td>User can specify which event the trace subsystem is sensitive to and what data is recorded</td>
</tr>
<tr>
<td>Profiler</td>
<td>all</td>
<td>Function and idle profile</td>
</tr>
<tr>
<td>Current Trace Subsystem</td>
<td>DCU3, DCU2</td>
<td>Read the current trace subsystem settings. If the trace subsystem is not currently configured then command language variables are used.</td>
</tr>
</tbody>
</table>

ST20 Embedded Software Toolset Training:
Trace post-mortem can be used to examine what was last recorded into the trace buffer before the application had crashed.

Typically when an application has crashed, then the debug session needs to be terminated.

It can then be restarted in post-mortem mode by selecting **Postmortem extraction** from the **Start Debug** menu available in the **Build** menu.
To open go to **Tools menu → Options → Postmortem settings tab.**
The trace buffer

- A trace buffer must be defined in the memory of the target.
- By default, 16 words are available in the TRAPHANDLER memory segment.
- However, a user may wish to locate the trace buffer somewhere else in memory, or utilise a larger trace buffer.
- Typically the user can reserve space by using a symbol that is linked into the application; i.e.

  ```c
  int tracebuff[256];
  ```

- Ideally for post-mortem debugging the trace buffer is located in the INTERNAL memory segment.
Exercise 6

In this exercise you will …

• Explore the Trace & Profile features of the debugger.
• The exercise uses a program that will fail in one of 9 different ways.
• Use the features of the debugger to identify the cause of each failure.
• In some cases the failure may cause the target to crash. Thus, the post-mortem debugging features will need to be explored.

See handout for detailed instructions
In this Session you have learned how to…

- Use the Trace & Profile features of the Tools.
- Be able to perform Post-mortem debugging.
- Customise the trace buffer.