Debugging Embedded Systems

Sergio Scaglia
March 2011
Agenda

- What is Debug?
- Debugging Embedded Systems
- GDB/MI Interface
- GDB Remote Serial Protocol
- JTAG Overview
- ARM Debug Architecture v5
- SWD Protocol
- Debug+ETM Connectors
- Debug Session Example
Debugging Embedded Systems

- What is Debug?
- Debugging Embedded Systems
- GDB/MI Interface
- GDB Remote Serial Protocol
What is Debug?

- Debug is a Science or an Art?

- The Principle of Confirmation:
  - Debug is the process of confirming, one by one, that the many things we believed (or assumed) to be true about the code, actually are true.
  - Finding one of the assumptions is not true, we found a bug.

- Debugging Techniques:
  - Adding Trace Code to the program to print out messages or values of variables, as the program executes (using `printf()` for example).
  - Using a Debugging Tool (such as GNU GDB).
Debuggers vs Front-Ends

- **Debuggers:**
  - Computer program used to test and debug other programs (the “Target” program). Example: GNU GDB

- **Front-Ends:**
  - Computer programs that monitor and control a Debugger program (those that only present a Command Line Interface), allowing the users to debug their programs using a GUI-based environment. Examples: DDD, Eclipse, Insight

- **Text-based vs GUI-based Tools**
  - In general, GUI-based tools are more convenient and efficient from the user point of view.
  - Front-Ends are limited to the back-end features.
  - In some cases (such as debugging GUI applications, quick checks) text-based tools are preferred (no clashes, lower startup time).
GNU GDB Screenshot

C:\\arm-none-eabi-gdb
GNU gdb (Sourcery G++ Lite 2009g3-60) 6.8.50.20090630-cvs
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=i686-mingw32 --target=arm-none-eabi".
For bug reporting instructions, please see:
<https://support.codesourcery.com/GNUToolchain/>.
(gdb) help
List of classes of commands:
aliases — Aliases of other commands
breakpoints — Making program stop at certain points
data — Examining data
files — Specifying and examining files
internals — Maintenance commands
obscure — Obscure features
running — Running the program
stack — Examining the stack
status — Status inquiries
support — Support facilities
tracepoints — Tracing of program execution without stopping the program
user-defined — User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Command name abbreviations are allowed if unambiguous.
(gdb)
DDD Screenshot
Eclipse (Debug perspective) Screenshot
Debugger: Main Operations

- Stepping Through the Source Code
  - Run
  - Next
  - Step (inside functions call)
  - Continue

- Breakpoints
  - Break
  - TBreak

- Inspecting Variables
  - Print
Debugger: Main Operations (Cont.)

- Detecting Changes in the Variables (Watchpoints)
  *(Breakpoint + Variable Inspection)*
  - Watch var
  - Watch (var > 100)

- Inspecting the Call Stack
  *(Stack stores function’s local variables, parameters, and location from which the function was called)*
  - Frame 1
  - Up / Down
  - BackTrace
Debugging Embedded Systems – Typical Scenario

- Eclipse (GDB Front-End)
- GDB Debugger
- Mi interface
- Remote Serial Protocol (TCP connection)
- GDB Server
- USB/Ethernet
- JTAG/SWD Protocol
- JTAG/SWD Emulator
- HOST
- TARGET
GDB/MI Interface

- Command Line Interface displays human-oriented messages.
- GDB/MI is a line based machine oriented text interface to GDB.
- It is specifically intended to support the development of systems which use the debugger as just one small component of a larger system (Front-Ends, IDEs, etc)
- CLI commands are still accepted.
- It’s activated using the –i mi command line option:
  - Example: arm-none-eabi-gdb.exe -i mi -x rom.gdb timer.elf

GDB/MI Input Syntax

- \[[ token \] "-" operation ( " " option )* [ " --" ] ( " " parameter )* \] nl
- Example: -stop
  - 0123-break-insert –h main
  - -break-list
  - -exec-continue
GDB/MI Output Syntax

```
output ->
  ( out-of-band-record )* [ result-record ] "(gdb)" nl
result-record ->
  [ token ] "^" result-class ( "," result )* nl
out-of-band-record ->
  async-record | stream-record
async-record ->
  exec-async-output | status-async-output | notify-async-output
exec-async-output ->
  [ token ] "*" async-output
status-async-output ->
  [ token ] "+" async-output
notify-async-output ->
  [ token ] "=" async-output
async-output ->
  async-class ( "," result )* nl
result-class ->
  "done" | "running" | "connected" | "error" | "exit"
```
GDB/MI Interface (Cont.)

GDB/MI Output Syntax (Cont.)

```
async-class ->
   "stopped" | others (where others will be added depending on the needs|this
               is still in development).

result -> variable "=" value
variable -> string
value -> const | tuple | list
const -> c-string
tuple -> "{}" | "{" result ( "," result )* "}" 
list -> "[]" | "[" value ( "," value )* "]" | "[" result ( "," result )* "]"
stream-record ->
   console-stream-output | target-stream-output | log-stream-output

console-stream-output ->
   "~" c-string

target-stream-output ->
   "@" c-string

log-stream-output ->
   "&" c-string

nl -> CR | CR-LF
token -> any sequence of digits.
```
GDB/MI Interface (Cont.)

GDB/MI Output Syntax (Cont.)

Notes:

- **status-async-output** contains on-going status information about the progress of a slow operation. It can be discarded. All status output is prefixed by `+`.

- **exec-async-output** contains asynchronous state change on the target (stopped, started, disappeared). All async output is prefixed by `*`.

- **notify-async-output** contains supplementary information that the client should handle (e.g., a new breakpoint information). All notify output is prefixed by `=`.

- **console-stream-output** is output that should be displayed as is in the console. It is the textual response to a CLI command. All the console output is prefixed by `~`.

- **target-stream-output** is the output produced by the target program. All the target output is prefixed by `@`.

- **log-stream-output** is output text coming from gdb's internals, for instance messages that should be displayed as part of an error log. All the log output is prefixed by `&`. 
GDB/MI Output Records

Result Records

The response to a GDB/MI command includes one of the following result indications:

"^done" [ "," results ]
The synchronous operation was successful, results are the return values.

"^running"
The asynchronous operation was successfully started. The target is running.

"^connected"
gdb has connected to a remote target.

"^error" "," c-string
The operation failed. The c-string contains the corresponding error message.

"^exit"
gdb has terminated.
GDB/MI Interface (Cont.)

GDB/MI Output Records (Cont.)

Stream Records

The gdb internally maintains a number of output streams: the console, the target, and the log. The output intended for each of these streams is funneled through the gdb/mi interface using stream records:

"~" string-output

The console output stream contains text that should be displayed in the CLI console window. It contains the textual responses to CLI commands.

"@" string-output

The target output stream contains any textual output from the running target. This is only present when GDB's event loop is truly synchronous, which is currently only the case for remote targets.

"&" string-output

The log stream contains debugging messages being produced by gdb's internals.
GDB/MI Interface (Cont.)

- GDB/MI Output Records (Cont.)
  - Async Records

They are used to notify the gdb/mi client of additional changes that have occurred. Those changes can either be a consequence of gdb/mi commands (e.g., a breakpoint modified) or a result of target activity (e.g., target stopped).

*running,thread-id="thread"

*stopped,reason="reason"

(some example of reasons; breakpoint-hit, watchpoint-trigger, function-finished, exited-normally, signal-received)

=thread-created,id="id"

=thread-exited,id="id"
GDB/MI Interface (Cont.)

GDB/MI Examples:

Setting a Breakpoint:

```
-> -break-insert main
<- ^done,bkpt={number="1",type="breakpoint",disp="keep", enabled="y",addr=0x08048564",func="main",file="myprog.c", fullname="/home/nickrob/myprog.c",line="68",times="0"}
<- (gdb)
```

Program Execution:

```
-> -exec-run
<- ^running
<- (gdb)
<- *stopped.reason="breakpoint-hit",disp="keep",bkptno="1",thread-id="0", frame={addr=0x08048564,func="main", args=[{name="argc",value="1"},{name="argv",value="0xbfc4d4d4"}], file="myprog.c",fullname="/home/nickrob/myprog.c",line="68"}
<- (gdb)
-> -exec-continue
<- ^running
<- (gdb)
<- *stopped.reason="exited-normally"
<- (gdb)
```
GDB/MI Interface (Cont.)

GDB/MI Examples: (Cont.)

- Bad Command:
  
  ```
  -> -rubbish
  <- `error,msg="Undefined MI command: rubbish"
  <- (gdb)
  ```

- Quitting GDB:

  ```
  -> (gdb)
  <- `-gdb-exit
  <- `exit
  ```
Debugging Embedded Systems – Typical Scenario

- Eclipse (GDB Front-End)
- GDB Debugger
- Mi interface
- Remote Serial Protocol (TCP connection)
- GDB Server
- USB/Ethernet
- JTAG/SWD Protocol
- JTAG/SWD Emulator
- HOST
- TARGET
GDB Remote Serial Protocol

- GDB Remote Serial Protocol allows GDB to connect to any target remotely.
- GDB implements the RSP client side, while the target side needs to implement the RSP Server (for example; GDB Server)
- Communication channel for RSP can be a Serial or TCP/IP connection.
- RSP Packet format:

```
$.............................. # h h
```

- **Packet Acknowledgement**
  - Each packet should be acknowledged with a single character:
    - ‘+’ to indicate satisfactory reply
    - ‘-’ to indicate failure and request retransmission

- **Interrupt**
  - GDB can interrupt the server transmitting character 0x03 between packets.
  - GDB Server can ignore the interrupts.
GDB Remote Serial Protocol (Cont.)

RSP Packets (Standard Remote Debug)

? Report why the target halted.

c, C, s and S. Continue or step the target (possibly with a particular signal). A minimal implementation may not support stepping or continuing with a signal.

D. Detach from the client.

g and G. Read or write general registers.

qC and H. Report the current thread or set the thread for subsequent operations. The significance of this will depend on whether the target supports threads.

k. Kill the target. The semantics of this are not clearly defined. Most targets should probably ignore it.

m and M. Read or write main memory.

p and P. Read or write a specific register.

qOffsets. Report the offsets to use when relocating downloaded code.

qSupported. Report the features supported by the RSP server. As a minimum, just the packet size can be reported.

qSymbol: (i.e. the qSymbol packet with no arguments). Request any symbol table data. A minimal implementation should request no data.

vCont?. Report what vCont actions are supported. A minimal implementation should return an empty packet to indicate no actions are supported.

X. Load binary data.

z and Z. Clear or set breakpoints or watchpoints.
GDB Remote Serial Protocol (Cont.)

RSP Packets in the **remote target** command

<table>
<thead>
<tr>
<th>GDB (RSP Client)</th>
<th>Target (RSP Server)</th>
</tr>
</thead>
<tbody>
<tr>
<td>qSupported</td>
<td>Report packet size supported</td>
</tr>
<tr>
<td>PacketSize-119</td>
<td></td>
</tr>
<tr>
<td>?</td>
<td></td>
</tr>
<tr>
<td>50S</td>
<td>Report we stopped due to signal 5 (TRAP exception)</td>
</tr>
<tr>
<td>Hc-1</td>
<td></td>
</tr>
<tr>
<td>OK</td>
<td>Future step/continue operations on all threads</td>
</tr>
<tr>
<td>qC</td>
<td></td>
</tr>
<tr>
<td>{empty}</td>
<td>Current thread is -1</td>
</tr>
<tr>
<td>qOffsets</td>
<td></td>
</tr>
<tr>
<td>Text=0;Data=0;Size=0;</td>
<td>Report offsets to be used when loading code</td>
</tr>
<tr>
<td>Hg-1</td>
<td></td>
</tr>
<tr>
<td>OK</td>
<td>All other future operations should apply to all threads</td>
</tr>
<tr>
<td>g</td>
<td></td>
</tr>
<tr>
<td>0000 ... 10000000</td>
<td>Report all general register values</td>
</tr>
<tr>
<td>qSymbol::</td>
<td>Offer to provide symbol values (none required)</td>
</tr>
<tr>
<td>OK</td>
<td></td>
</tr>
</tbody>
</table>

(gdb) target remote :51000
Remote debugging using :51000
0x000000100 in _start ()
(gdb)
GDB Remote Serial Protocol (Cont.)

- RSP Packets in the `load` command

<table>
<thead>
<tr>
<th>GDB (RSP Client)</th>
<th>Target (RSP Server)</th>
</tr>
</thead>
<tbody>
<tr>
<td>X0,0:</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td>Empty load to test if binary write is supported</td>
</tr>
<tr>
<td>X0,100:&lt;binary data&gt;</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td>Load first 256 bytes of text section</td>
</tr>
<tr>
<td>X1200,90:&lt;binary data&gt;</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td>Load final block of text section</td>
</tr>
<tr>
<td>X0,e:&lt;binary data&gt;</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td>Load data section</td>
</tr>
<tr>
<td>P21=00000100</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td>Set program counter to start of loaded code</td>
</tr>
</tbody>
</table>

(gdb) load hello
Loading section .text, size 0x1290 lma 0x0
Loading section .rodata, size 0xe lma 0x1290
Start address 0x100, load size 4766
Transfer rate: 5 KB/sec, 238 bytes/write.
(gdb)
GDB Remote Serial Protocol (Cont.)

RSP Packets in the **step** command

<table>
<thead>
<tr>
<th>GDB (RSP Client)</th>
<th>Target (RSP Server)</th>
</tr>
</thead>
<tbody>
<tr>
<td>m0, 4</td>
<td>Word (instruction) at previous program counter address</td>
</tr>
<tr>
<td>0000000</td>
<td></td>
</tr>
<tr>
<td>s</td>
<td></td>
</tr>
<tr>
<td>s05</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
</tr>
<tr>
<td>0000 ... 0008411</td>
<td></td>
</tr>
<tr>
<td>s</td>
<td></td>
</tr>
<tr>
<td>s05</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
</tr>
<tr>
<td>0000 ... 0008011</td>
<td></td>
</tr>
<tr>
<td>m1254, 4</td>
<td></td>
</tr>
<tr>
<td>9c21fff4</td>
<td></td>
</tr>
<tr>
<td>m1258, 4</td>
<td></td>
</tr>
<tr>
<td>d4011004</td>
<td></td>
</tr>
<tr>
<td>m7cfd4, 4</td>
<td></td>
</tr>
<tr>
<td>000000118</td>
<td></td>
</tr>
</tbody>
</table>

Single step. Return control with signal 5 (TRAP)

Report all general register values

Single step. Return control with signal 5 (TRAP)

Report all general register values

First function prologue instruction

Second function prologue instruction

Return address within current stack frame

(gdb) step
main () at hello.c:41
41  simputs( "Hello World!\n" );
(gdb)
GDB Remote Serial Protocol (Cont.)

- RSP session capture, showing the $S$ command:
Debugging Embedded Systems

- JTAG Overview
- ARM Debug Architecture v5
- SWD Protocol
- Debug+ETM Connectors
Debugging Embedded Systems – Typical Scenario

- Eclipse (GDB Front-End)
- GDB Debugger
- Mi interface
- Remote Serial Protocol (TCP connection)
- GDB Server
- USB/Ethernet
- JTAG/SWD Protocol
- JTAG/SWD Emulator
- HOST
- TARGET

NXP
IEEE 1149.1 (JTAG) Overview

- Originally designed for Boundary Scan Tests (used for testing IC’s interconnects at PCB level)
- Also used to access sub-blocks of integrated circuits and to debug Embedded Systems.
IEEE 1149.1 (JTAG) Overview

- The JTAG ecosystem begins with IC designers embedding test logic in each chip and connecting internal registers in the chip to JTAG scan chains. The hardware components of IEEE 1149.1 consist of:

  - **TAPs.** Four mandatory pins are: TDI (test data in), TDO (test data out), TMS (test mode select), and TCK (test clock). An optional TRST/reset pin is also defined. When driven low, it resets the internal state machine.

  - **TAP controller:** A finite state machine with 16 states with TMS and TCK as its inputs. Outputs include ClockDR, UpdateDR, shiftDR, ClockIR, UpdateIR, ShiftIR, Select, Enable, TCK, and the optional TRST.

  - **Instruction register.**

  - **Test data register.**

  Test engineers use these structures as the access points for built-in self-test (BIST). Together, JTAG and BIST are widely used to deploy low overhead embedded test solutions that detect static faults such as shorts, opens, and logic errors.

    - For software debugging, design engineers use an in-circuit emulator to access an on-chip debug module, which is integrated into the CPU over the JTAG interface. This debug module provides software developers the ability to load, run, halt, and step the CPU.
JTAG TAP Controller State-Machine

The Test Access Port Controller

- 16-state TAP provides 4 major operations:
  - RESET
  - RUN-TEST
  - SCAN-DR
  - SCAN-IR

- Scans consist of 3 primary steps:
  - CAPTURE
  - SHIFT
  - UPDATE
## JTAG Interface signals

### JTAG TAP Interface Signals

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCK</td>
<td>Test Clock</td>
<td>Synchronizes the internal state machine operations</td>
</tr>
<tr>
<td>TMS</td>
<td>Test Mode State</td>
<td>Sampled at the rising edge of TCK to determine the next state</td>
</tr>
<tr>
<td>TDI</td>
<td>Test Data In</td>
<td>Represents the data shifted into the device's test or programming logic. It is sampled at the rising edge of TCK when the internal state machine is in the correct state.</td>
</tr>
<tr>
<td>TDO</td>
<td>Test Data Out</td>
<td>Represents the data shifted out of the device's test or programming logic and is valid on the falling edge of TCK when the internal state machine is in the correct state.</td>
</tr>
<tr>
<td>TRST</td>
<td>Test Reset</td>
<td>An optional pin which, when available, can reset the TAP controller's state machine</td>
</tr>
</tbody>
</table>
JTAG Shortcomings

- Not specifically designed for Debugging Embedded Systems
- The 4-pin connector makes it inconvenient for some low pin-count packages.
- Daisy-chaining is intolerant to a debug component (especially when a TAP is removed as a result of power management)

Daisy-chain debug topology
Serial Wire Debug – A new approach

To address the issues with JTAG, a new packet-based protocol was developed: Serial Wire Debug (SWD).

ARM designed a new debug architecture (CoreSight) taking into account the requirements for Debug and Trace. SWD was designed with this new debug architecture in mind.
ARM CoreSight Debug Architecture Overview

- Bus-based approach (instead of the serial scan interface) for on-chip debug control and access. Advantages:
  - It permits debug access while the CPU is running.
  - Debug access is more abstract, less CPU-specific and more scalable.
- Provides advantage for SoCs that consist on multiple IP blocks from multiple vendors, often using multiple clock and power domains.
- Register-based access to debug configuration and status information.
- CPU can also access this registers, giving additional flexibility.
- Star Topology of the bus, providing true power and clock isolation.
- New ARM Debug Interface v5 (ADIv5) implements CoreSight architecture.
- The Debug Access Port (DAP) decouples the external debug interface (JTAG or SWD) from the internal bus-based debug and trace infrastructure.
The DAP bus has a single master (Debug Port) and one or more Slaves (Access Ports) which are typically used to interface to the various on-chip bus standards, or to provide dedicated debug features.
Debug Access Port components

- AP (Access Ports) which can access various slaves devices:
  - JTAG-AP: legacy JTAG cores
  - MEM-AP for system memory (for example, the AHB-AP for Advanced High-performance Bus)
  - MEM-AP for bus-based debug functionality (for example, APB-AP for Advanced Peripheral Bus)
  - Other debug or control devices.

- Serial Wire/JTAG Debug Port (SWJ-DP)
  - It provides a mechanism to select between JTAG and SWD interfaces
  - Data signal SWDIO and clock SWCLK can be overlaid on the JTAG TMS and TCK pins (thus, both SWD and JTAG can share same connector)
SWJ-DP

- JTAG is selected by default, enabling legacy devices to continue working as normal.
- SWD protocol is selected by sending a specific sequence on TMS (switching protocol).
Debug Access Port accesses description

- **DAP accesses**
  - DAP bus has a single master (Debug Port) and one or more Slaves (Access Ports) which are typically used to interface to the various on-chip bus standards, or to provide dedicated debug features.
  - Each transaction sent from the external debugger is addressed to a single one of these components (the DP or an AP).

- **AP accesses**
  - Each AP provides up to 64 32-bit registers (arranged in 16 banks of 4).
  - One register identifies the particular AP type.
  - In a MEM-AP, a couple of registers are used for Address and Data.

- **DP accesses**
  - Registers in the DP are used to provide control and status of the external link (SWD).
  - The DP also has registers to select the current AP, and the register bank within that AP.

  *Note: Sticky bits are used to signal when some error conditions occur.*
Implementation Example: Debug Access Port (JTAG-DP accesses to an AP)
Breakpoints: Hardware vs Software

- **Hardware Instructions Breakpoints (requires specific hardware support)**
  - Provided by comparators called Watchpoint units
  - Watchpoint units can be programmed to detect data accesses and instruction fetches by monitoring Address bus, Data bus and control signals.
  - When a match occurs, the core is forced into Debug state.
Breakpoints support

- Software Instructions Breakpoints
  - It replaces the instruction code where the Breakpoint is set, by an special instruction (SW Interrupt) which causes the processor to attend a special handler routine (reserved for Debug).
  - Only can be used when running from code from RAM
  - No limit (in theory) in the number of software breakpoints that can be used.
Debugging Embedded Systems – Typical Scenario

- **Eclipse (GDB Front-End)**
- **GDB Debugger**
- **Mi interface**
- **Remote Serial Protocol (TCP connection)**
- **USB/Ethernet**
- **JTAG/SWD Protocol**
- **JTAG/SWD Emulator**
- **HOST**
- **TARGET**
SWD Protocol

- Each (successful) SWD transfer consists of 3 parts:
  - A Header (sent from the Host)
  - An Acknowledgement from the Target (if it recognizes the header)
  - A Data Payload (provided by the Host in a Write, or by the Target in a Read)
SWD Protocol – Header fields

Header fields description:

- **Start**: A single start bit, with value 1.
- **APnDP**: A single bit, indicating if DP or AP is accessed (0=DP, 1=AP)
- **RnW**: A single bit, indicating Read or a Write access (0=Write, 1=Read).
- **A[2:3]**: Two bits used to address internal register in DP or AP (transmitted LSB first on the wire).
- **Parity**: A single parity bit (even parity).
- **Stop**: A single stop bit, with value 0.
- **Park**: A single bit where the Host does not drive the line. The line is pulled High by the SWD interface hardware. Target reads this bit as 1.
- **Trn**: Turnaround. The line is not driven and the state of the line is undefined. The length is controlled by the TURNROUND field in the Wire Control Register. Default value; one clock cycle.
SWD Protocol – Acknowledgement and Data fields

- **Acknowledgement field:**
  - **ACK[0:2]:** A 3-bit target-to-host response (transmitted LSB first on the wire)
    - **100:** OK response
    - **010:** WAIT response
    - **001:** FAULT response

- **Data Payload:**
  - **WDATA[0:31]:** 32 bits of write data, from host to target (transmitted LSB first on the wire).
  - **RDATA[0:31]:** 32 bits of read data, from target to host (transmitted LSB first on the wire).
  - **Parity:** A single even-parity bit calculated over the data bits.
SWD Protocol – Timing relationship

- SWDIO and SWCLK timing relationship in a Read transfer:
SWD – Transfer examples

- Successful Write Operation (OK response)

- Successful Read Operation (OK response)
SWD – Transfer examples

- WAIT response to Read or Write Operation

- FAULT response to Read or Write Operation
A 20-pin IDC connector is used for Debug.
A 38-pin MICTOR connector is used for Trace.
Connector Types – Cortex Debug connector

- A 10-pin connector is used for Cortex Debug
Connector Types – Cortex Debug+ETM connector

- A 20-pin connector is used for Debug+Trace
Debugging Embedded Systems – Typical Scenario

- Eclipse (GDB Front-End)
- GDB Debugger
- GDB Server
- USB/Ethernet
- JTAG/SWD Emulator
- Mi interface
- Remote Serial Protocol (TCP connection)
- JTAG/SWD Protocol
- HOST
- TARGET
Debugging Embedded Systems

- Debug Session Example