Essential ARM Cortex-M Debugging with GDB

Kevin Townsend
Linaro

Linaro Connect
VIRTUAL • SPRING 2021

Overview

- Core Concepts
  - Connecting
  - Basic Navigation
  - Breakpoints
  - Watchpoints
  - Contextual Information
  - Stack Backtrace
  - Printing
  - Examining Memory
  - Examining Source Code
  - Searching Memory
  - Multiple Image Support

- Practical Examples
  - ISR Stack Rollback
  - Fault Identification

- Python Scriptability
  - Core GDB Functions
  - Minimal Python GDB Command
  - Loading Python GDB Commands
  - ARMv8-M (Cortex M33) Fault Handler

Core Concepts

Connecting

In one terminal window start GDB server:

```
   $ pyocd gdbserver --target=lpc55s69
or $ JLinkGDBServer -if SWD          \
     -speed 4000 -USB -device lpc55s69
```

Various tools like `pyocd` and `JLinkGDBServer` can be used to start a new GDB server. Parameters and requirements vary by tool, target hardware and debugger.

Then connect to it in a second window:

```
$ arm-none-eabi-gdb-py               \
    -s build/zephyr/zephyr.elf       \
    -ex "target remote tcp:localhost:3333"
```

Connect to the server to start a new (local or remote) debug session over TCP. Check the GDB Server output for the correct TCP port!

```
(gdb) monitor reset halt
(gdb) break main
(gdb) continue
```

Reset the firmware and stop after reset
Set a breakpoint at `main()`
Restart execution

Basic Navigation

ctrl+c	# Halt current program execution
c/continue	# Resume execution
s/step	# Step into function
s 10	# Step next 10 sources lines
n/next	# Run next line in func (step over)
n 10	# Run next 10 lines in current func
u/until 20	# Run until line 20 of current file
f/finish	# Run to the end of func/stack frame

💡 **`c/continue`** format with a ‘/’ indicates the shortcut command and the full command in this presentation. Either of the two values can be used to the same effect.

Breakpoints (fast!)

b/break main	# Break on main() entry
b main.c:func	# Break on func() in main.c
b main.c:18	# Break on line 18 of main.c
b main.c:18 if foo > 20	# Break if ‘foo’ > 20 (boolean cond)
tbreak main	# Fires once, deletes itself
info breakpoints	# List all breakpoints
ignore 2 20 ⭐	# Ignore bp 2 the first 20 hits
disable 2	# Disable bp 2
delete 2	# Delete bp 2

Watchpoints (Powerful, but slooooow)

Execution halts when variable is accessed or modified

watch foo	# Watch foo
watch myarray[10].val	# Watch .val in myarray[10]
watch *0x1000FEFE	# Watch memory addr 0x1000FEFE
watch foo if foo > 20 ⭐	# Conditional watch (foo > 20)
watch foo if foo + x > 20	# Complex conditional expression
info watchpoints	# List watchpoints
delete 7	# Delete watchpoint 7

Contextual Information

● info locals	# Local variables
● info variables	# Global variables
● info args	# Function argument variables
● info registers	# Core registers

Stack Backtrace

bt	# Display a stack backtrace (function call history)
frame	# Display the current stack frame
up	# Move up the stack (to main)
down	# Move down (away from main)

Printing

`p/print [/FMT] expression`

a (address)
c (char)
d (decimal int)
f (float)
o (octal int)
t (binary int)
u (unsigned decimal int)
x (hex int)

`p foo`	# Print the value of ‘foo’
`p foo+bar`	# Print complex expression
`p/x &main`	# Print address of main()
`p/x $r4`	# Print register R4 in hex
`p/a *(uint32_t[8]*)0x1234` ⭐	# Print array of 8 u32s @ 0x1234

Examining Memory

x [/FMT] addr

FMT is a **repeat count**, followed by a format and size letter.

x (hex)
d (decimal)
u (unsigned dec.)
t (binary)
f (float)
a (address)
i (instruction)
c (char)
s (string)
z (padded hex)

b (byte, 1b)
h (halfworld, 2b)
w (word, 4b)
g (giant, 8b)

x foo	# Show address of variable foo
x/4c 0x581F	# Show four chars @ 0x581F
x/4xw &main	# Show four words in hex @ main()

Examining Source Code

- list
- list *0x1234
- list main.c:func

- disas func

Show src for current location

Show src at the address 0x1234

Show src for func() from main.c

List ASM code for func()

Searching Memory

-
  ```
  find /b 0x0, 0x10000, 'H', 'e', 'l', 'l', 'o'
          0x581f
          1 pattern found.
  ```

  ```
  # Search for a byte pattern
    between 0x0 and 0x10000
  ```

-
  ```
  x/s 0x581f
          0x581f:    "Hello World! %s\n”
  ```

  ```
  # Examine string @ 0x581F
  ```

💡 Useful when checking for stack overflow, if stack memory is pre-filled with a known pattern.

Multiple Image Support

GDB parses one ELF file at a time for symbol lookup.

Complex projects may have two or more files. For example:

1. bootloader.elf
2. trustzone_secure.elf (TF-M)
3. trustzone_nonsecure.elf (Zephyr)

The ELF file can be switched while debugging via:

- symbol-file trustzone_secure.elf # Load new elf file

Practical Examples

ISR Stack Rollback (ARMv7-M, ARMv8-M)

p/t $lr	# Display binary value of `lr` register If b2 = 1, use `psp` If b2 = 0, use `msp`
p/a *(uint32_t[8] *)$psp or p/a *(uint32_t[8] *)$msp	# Display the `psp` or `msp` stack frame. 7th value is the `pc` register value
list *0x12345678	# Show source for `pc` addr

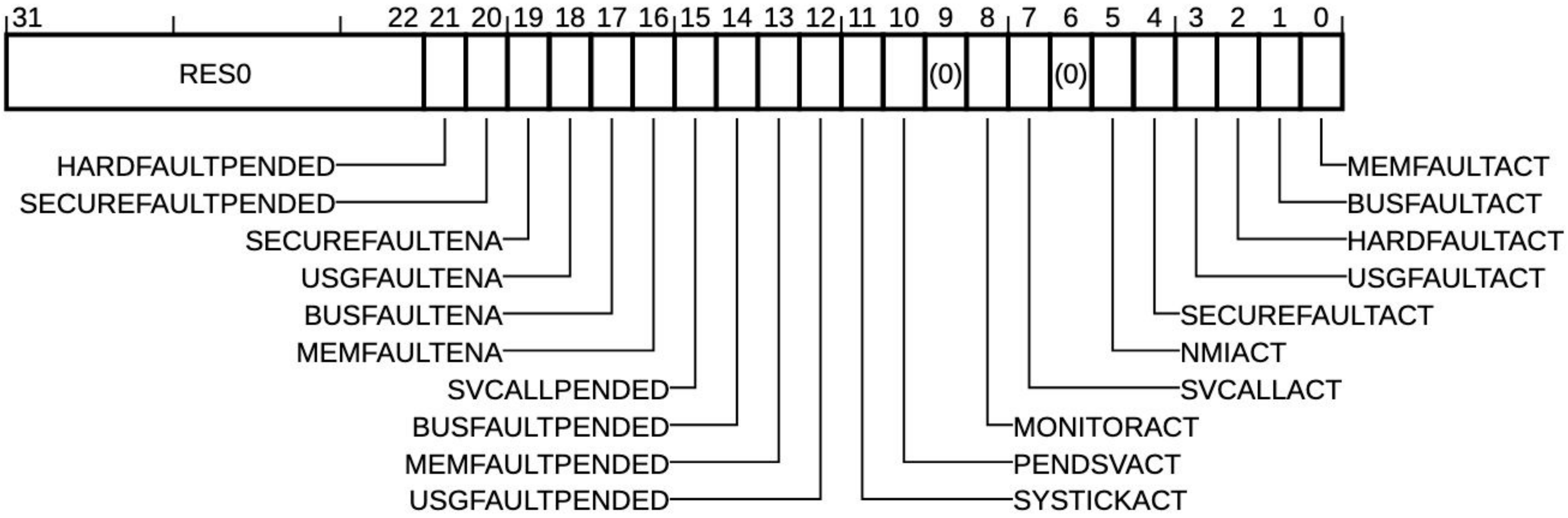
Fault Identification (ARMv8-M)

- p/x *(uint32_t*)0xE000ED24 # System Handler Control & State Register (SHCSR)
- p/x *(uint32_t*)0xE000ED2C # HardFault Status Register (HFSR)
- p/x *(uint32_t*)0xE000ED28 # Configurable Fault Status Register (CFSR)

❗ JLinkGDBServer (V6.86) was used to access this memory range. Pyocd 0.30.0 seems to have access restrictions to this memory range, further investigation required as to why.

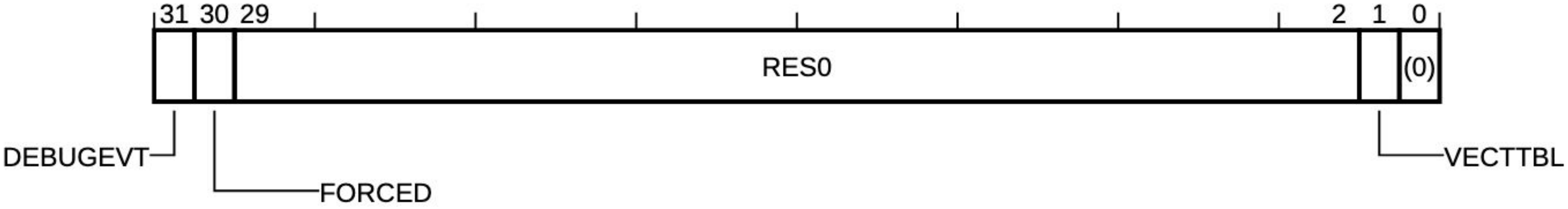
SHCSR

System Handler Control & State Register (0xE000ED24)



HFSR

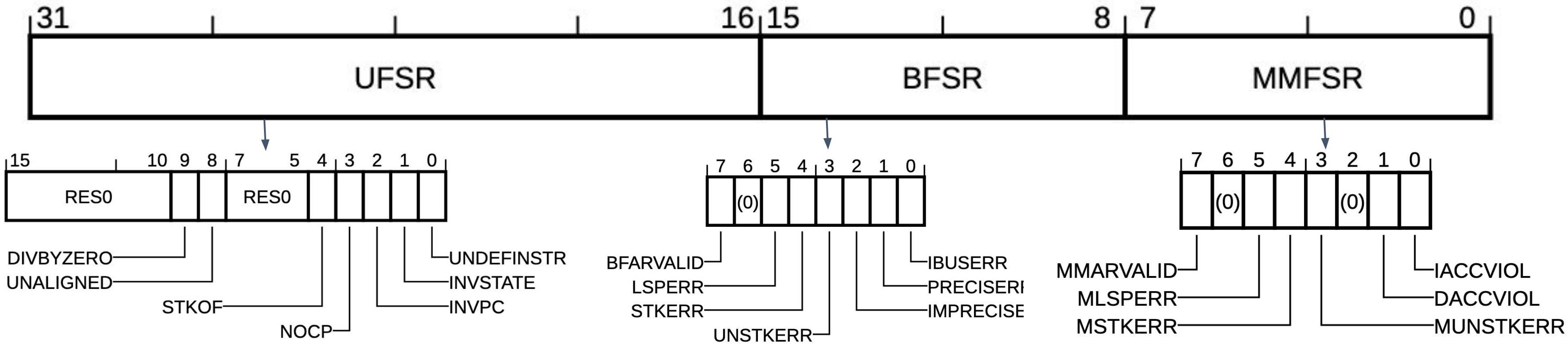
HardFault Status Register (0xE000ED2C)



- VECTTLB = 1 means a vector table read fault has occurred
- FORCED = 1 means the processor has escalated a configurable-priority exception to HardFault

CFSR

Configurable Fault Status Register (0xE000ED28)



- MMFSR = MemManage Fault Status Register
- BFSR = BusFault Status Register
- UFSR = UsageFault Status Register

Fault Identification: Example

-
  ```
  (gdb) p/x *(uint32_t *)0xE000ED24
  $4 = 0x50008
  ```

-
  ```
  (gdb) p/x *(uint32_t *)0xE000ED28
  $6 = 0x200
  ```

SHCSR = 0x50008 indicates that we
have a **UsageFault**

CFSR bits 16-31 = UFSR
Divide by zero

Python Scriptability

Python GDB Integration

Recent versions of GDB allows us to extend the GDB server with custom commands or helper functions written in Python. Functions must be loaded into the GDB session.

You interact with GDB in python via gdb.*, for example:

```
# Read the system handler control & state register
shcsr = int(gdb.parse_and_eval("*(uint32_t *)0xE000ED24"))
print("SHCSR: 0x%08X" % shcsr)
```

❗ Not every version of GDB supports python, but many modern toolchains do. Some toolchains include Python and non-Python variants, such as the GNU Arm Embedded Toolchain with `arm-none-eabi-gdb` and `*gdb-py`.

Minimal Python GDB Command

```
#!/usr/bin/env python
import gdb

class Minimal (gdb.Command):
    def __init__(self):
    super(Minimal, self).__init__("minimal", gdb.COMMAND_USER)

    def invoke(self, arg, from_tty):
    # Get the current value of the PC register
    pc = gdb.parse_and_eval("$pc")
    print("PC: 0x%08X" % pc)

Minimal()
```

Command Name in GDB

Source available at:
https://bit.ly/3vrtkn6

Loading Python GDB Commands

- (gdb) source minimal.py # Load python script into GDB
- (gdb) minimal() # Execute new ‘`minimal`’ command
    PC: 0x10001C2C

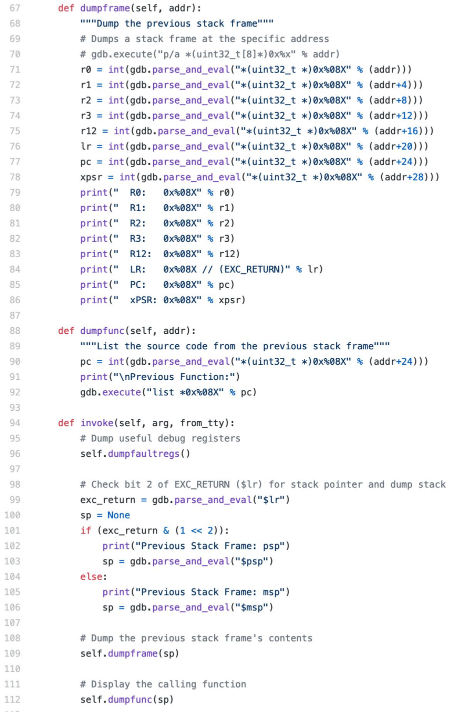
You can load a variety of commands in a single debug session.

GDB Python commands are useful to encapsulate complicated tasks involving numerous registers.

Cortex M33 (ARMv8-M) Fault Handler

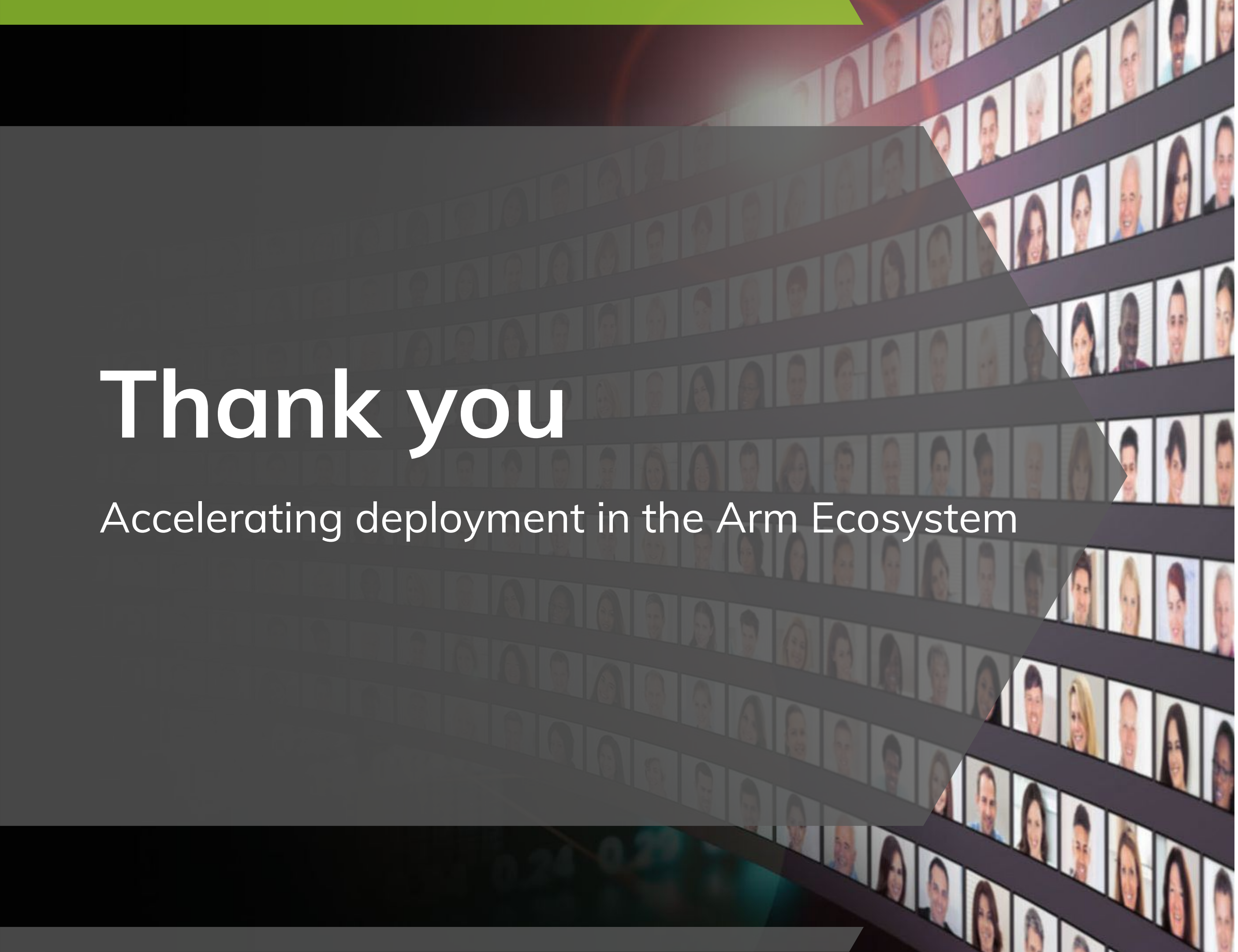
```
(gdb) faultdetails()
Fault Status Registers:
  SHCSR: 0x000F0008
  CFSR:  0x00000000
  HFSR:  0x00000000
  UsageFault exception active!
  UFSR: 0x0000

Previous Stack Frame: psp
  R0:   0x00000013
  R1:   0x0000000A
  R2:   0x80000000
  R3:   0x10001821
  R12:  0x00000000
  LR:   0x1000047D // (EXC_RETURN)
  PC:   0x1000047C
```

Source available at: https://bit.ly/3vrtkn6



```python
    def dumpframe(self, addr):
        """Dump the previous stack frame"""
        # Dumps a stack frame at the specific address
        # gdb.execute("p/a *(uint32_t[8]*)0x%x" % addr)
        r0 = int(gdb.parse_and_eval("*(uint32_t *)0x%08X" % (addr)))
        r1 = int(gdb.parse_and_eval("*(uint32_t *)0x%08X" % (addr+4)))
        r2 = int(gdb.parse_and_eval("*(uint32_t *)0x%08X" % (addr+8)))
        r3 = int(gdb.parse_and_eval("*(uint32_t *)0x%08X" % (addr+12)))
        r12 = int(gdb.parse_and_eval("*(uint32_t *)0x%08X" % (addr+16)))
        lr = int(gdb.parse_and_eval("*(uint32_t *)0x%08X" % (addr+20)))
        pc = int(gdb.parse_and_eval("*(uint32_t *)0x%08X" % (addr+24)))
        xpsr = int(gdb.parse_and_eval("*(uint32_t *)0x%08X" % (addr+28)))
        print("  R0:   0x%08X" % r0)
        print("  R1:   0x%08X" % r1)
        print("  R2:   0x%08X" % r2)
        print("  R3:   0x%08X" % r3)
        print("  R12:  0x%08X" % r12)
        print("  LR:   0x%08X // (EXC_RETURN)" % lr)
        print("  PC:   0x%08X" % pc)
        print("  xPSR: 0x%08X" % xpsr)

    def dumpfunc(self, addr):
        """List the source code from the previous stack frame"""
        pc = int(gdb.parse_and_eval("*(uint32_t *)0x%08X" % (addr+24)))
        print("\nPrevious Function:")
        gdb.execute("list *0x%08X" % pc)

    def invoke(self, arg, from_tty):
        # Dump useful debug registers
        self.dumpfaultregs()

        # Check bit 2 of EXC_RETURN ($lr) for stack pointer and dump stack
        exc_return = gdb.parse_and_eval("$lr")
        sp = None
        if (exc_return & (1 << 2)):
            print("Previous Stack Frame: psp")
            sp = gdb.parse_and_eval("$psp")
        else:
            print("Previous Stack Frame: msp")
            sp = gdb.parse_and_eval("$msp")

        # Dump the previous stack frame's contents
        self.dumpframe(sp)

        # Display the calling function
        self.dumpfunc(sp)
```

Linaro Connect
VIRTUAL • SPRING 2021



Thank you

Accelerating deployment in the Arm Ecosystem

Linaro Connect
VIRTUAL • SPRING 2021