Anti-debugging tricks for embedded devices

GreHack

8-bit_edition

Back in my day...

- On PCs, many ways to implement anti-debugging
  - IsDebuggerPresent() / MeltICE / …
- Was fun to look at and circumvent at the time

Asm Code
This is the full version of W32Dasm and is not to be copied or distributed as shareware.
MMX
Compatible
W 32 16 Dasm
32/16 Bit Disassembler/32 Bit Debugger
Version 8.9
Copyright 1996 URSoftware

Intro

- Access to debug interface usually means game over for a target device
- For a reason I cannot understand, most devices still keep this interface wide open
  - Even if the MCU allows to disable it
- Is there a way to prevent such access by software only ?

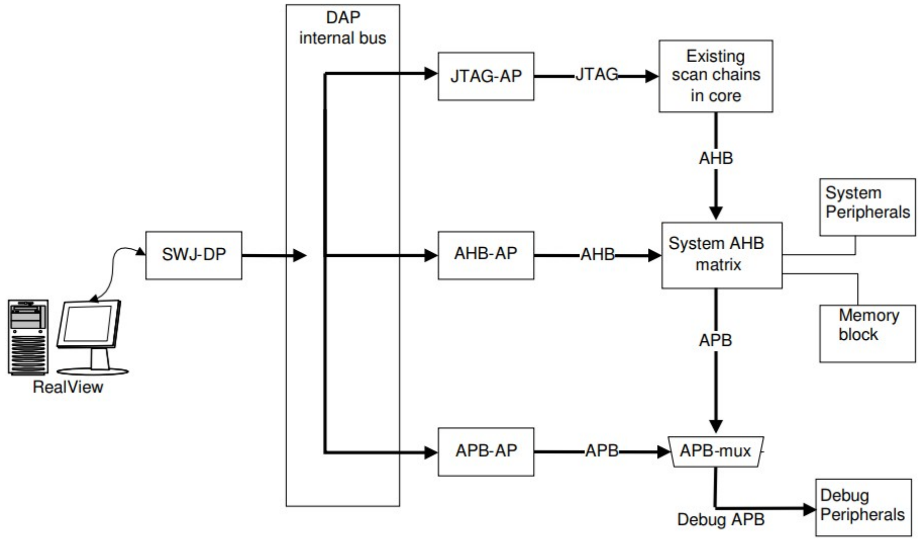
SWD – ARM’s debug interface

SWD

- Serial Wire Debug
  - Created by ARM as a replacement for JTAG
  - 2 main pins (CLK, IO)
- Used to access the DAP – Debug Access Port
  - Internal bus used for debugging purposes

Internal architecture

- DP – Debug Port
  - « Gateway » between interface and the DAP
- AP – Access Ports
  - Connected to the DAP
  - Each AP has a specific purpose
  - Similar to JTAG TAPs



Transactions

- AP queries are made through the DP
  - Write the *SELECT* register to select an AP
    - *APSEL* – AP address on the bus
    - *APBANKSEL* – AP register to select
  - Write or read *RDBUFF* register to access the AP register

Raw SWD interaction

- Reimplemented low-level SWD protocol for Hydrabus
- Python module
  - Low level protocol
    - headers, parity bit, status bits, turnover, …
  - High level functions
    - Read / Write to DP / AP

Demo

- pyHydrabus

Debugging and anti-debugging

MEM-AP

- Standard cell provided by ARM
  - Common to all Cortex- cores
- Allows to read and write to all AHB memory
  - Set *TAR* register with the address to access
  - Read or write *DRW* register to read/write from/to that address

Read / Write registers

- Again, simple « memory » transfers :
  - Select register with *DCRSR* (0xE000EDF4) register
  - Read/write from/to *DCRDR* (0xE000EDF8) register

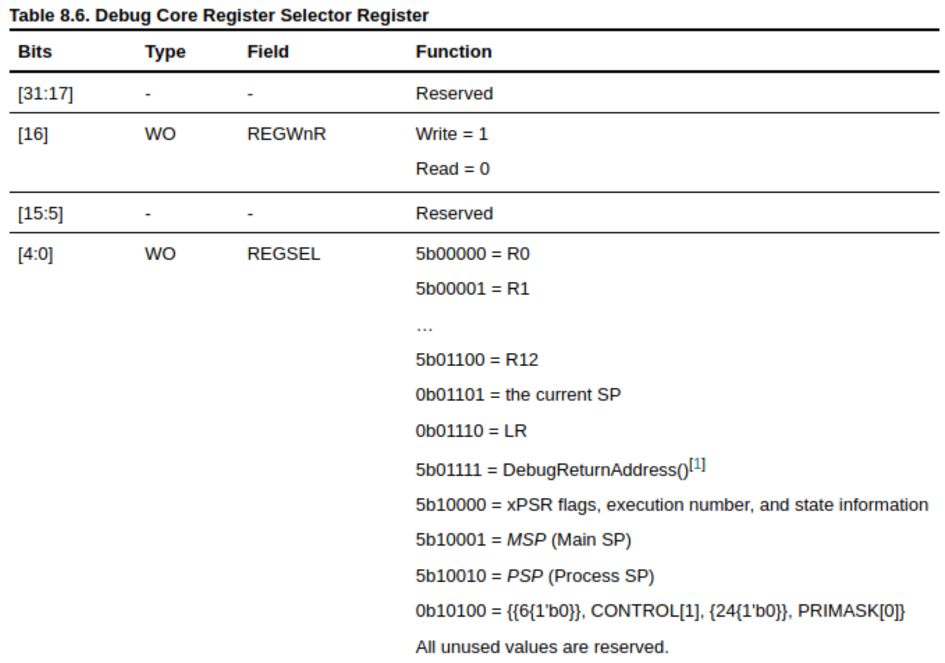


Table 8.6. Debug Core Register Selector Register

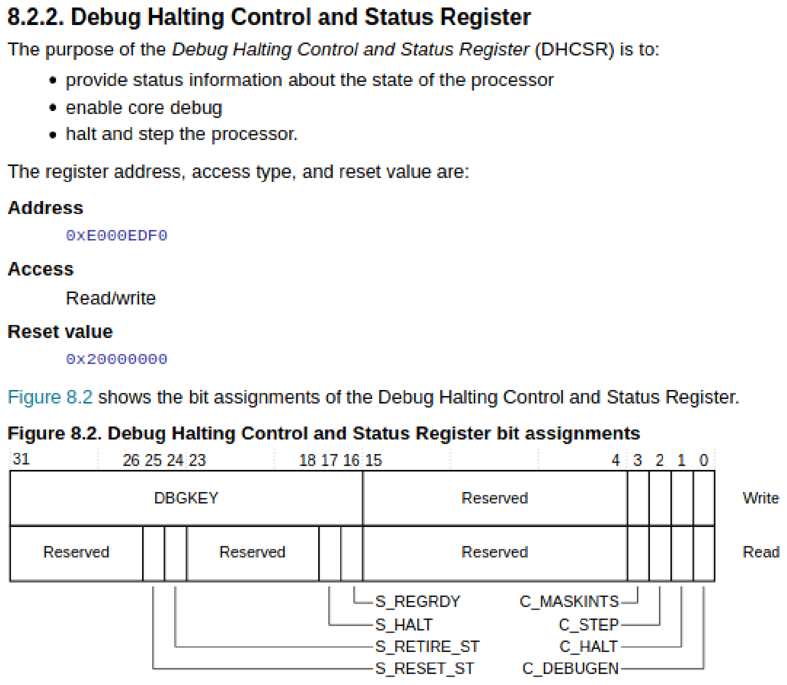
Bits	Type	Field	Function
[31:17]	-	-	Reserved

[16]	WO	REGWnR	Write = 1
			Read = 0

[15:5]	-	-	Reserved
[4:0]	WO	REGSEL	5b00000 = R0 5b00001 = R1 … 5b01100 = R12 0b01101 = the current SP 0b01110 = LR 5b01111 = DebugReturnAddress()[1] 5b10000 = xPSR flags, execution number, and state information 5b10001 = *MSP* (Main SP) 5b10010 = *PSP* (Process SP) 0b10100 = {{6{1'b0}}, CONTROL[1], {24{1'b0}}, PRIMASK[0]} All unused values are reserved.

MEM-AP usage

- The MEM-AP is not allowed to query the system memory unless the *C_DEBUGEN* bit in the *DHCSR* register is set
- Must set this bit before querying the MEM-AP



8.2.2. Debug Halting Control and Status Register

The purpose of the *Debug Halting Control and Status Register* (DHCSR) is to:

- provide status information about the state of the processor
- enable core debug
- halt and step the processor.

The register address, access type, and reset value are:

Address

`0xE000EDF0`

Access

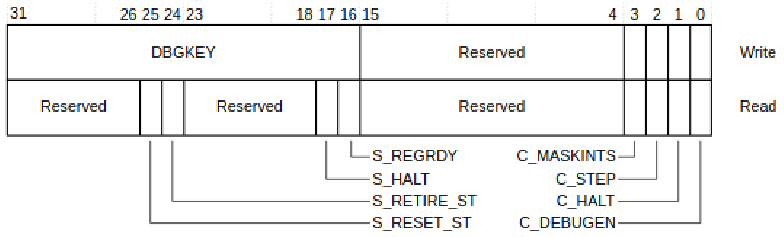
Read/write

Reset value

`0x20000000`

Figure 8.2 shows the bit assignments of the Debug Halting Control and Status Register.

Figure 8.2. Debug Halting Control and Status Register bit assignments



Detect debug access

- Querying this bit is easily achievable
- Cannot be written by the core
  - cannot disable it at runtime :(
- Does not work on Cortex-M0

```
void detect_debug(void)
{
  uint32_t *DHCSR=(uint32_t *) 0xE000EDF0;
  while(1) {
    if(*DHCSR&1) {
      printf("Debugger detected !\r\n");
      NVIC_SystemReset();
    }
  }
}
```

Demo

Bypassing detection

- OpenOCD (as others) first update *DHCSR* with *C_DEBUGEN* before updating it again with *S_HALT*
  - Leaves some time to detect debug access
- Setting both bits in *DHCSR* allows to halt the CPU as soon as the debug is requested.

Breakpoints

- The BPU (Breakpoint Unit) manages the hardware breakpoints
  - Cortex M0/M1, more on M3+ later
- If an enabled comparator matches PC address, the CPU is halted
- Query *DBGBCR* register
  - If *ENABLE* is set, a debugger should be present

Demo

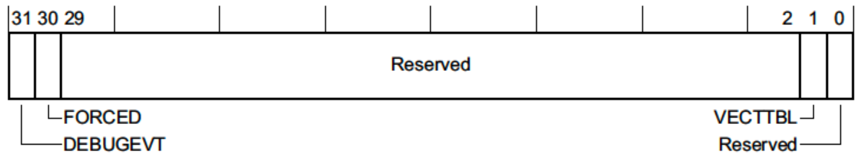
Bkpt instruction

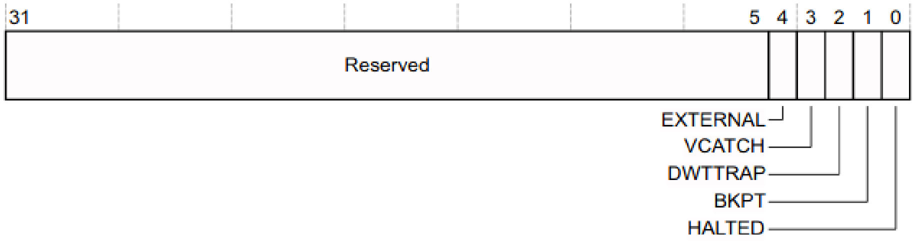
- The *bkpt* instruction is used to send a message to the debugger
- Has to be handled by the debugger
- If no debugger is present, the instruction will generate a HardFault

Custom HardFault handler

- Check the HFSR (M3+) or DFSR(M0) registers
- If DEBUGEVT/BKPT bit is set to 1, the Hardfault was triggered by a *bkpt* instruction

The HFSR bit assignments are:





Recovering a HardFault

- Registers are saved on the stack before calling the handler
  - R0, r1, r2, r3, r12, LR, PC, xPSR
- Execution state is set to handler mode.
  - Set LR to 0xfffffff9 and perform a *bx lr* to restore execution in thread mode
- Execution will resume from the faulty instruction

Demo

How to use these tricks?

Timers

- Use a timer IRQ to perform a periodic check
- Update a dedicated flag whenever a debugger is triggered

Threads

- If using a RTOS, create a dedicated thread that checks for a debugger
- Used this technique at Insomni’hack 2019
  - https://research.kudelskisecurity.com/2019/04/11/physically-unclonable-functions-in-practice/

Opaque predicates

- Detect debugger, and use the resulting value as part of a computation
  - Result will be different when debugging and not
- Way harder to reverse engineer

Anti-reversing

FPU

- Flash Patch & Breakpoint Unit
- FPB is used to implement hardware breakpoints
- Available on Cortex-M3/M4/M33, replaces BPU

FPB remap

- The FPB can also be used to remap an instruction to another one
- Useful to quickly fix an issue during debugging

Use FPB as obfuscation

- Use the FPB remap at runtime to change critical functions behavior
- Makes static firmware analysis way more complicated

Demo

Semihosting

Semihosting

- Semihosting is a way to communicate with a debugger through MEM-AP queries
- Uses a similar syntax to syscalls
  - open / seek / read / write / close
- Allows data to go from/to the debugger
- Can use *printf()* commands and get output directly in the debugger

Semihosting in details

- When in debugging mode, the core issues a service request 0xAB
- The debugger traps this ISR, fetches the operation number in r0 and parameters in r1
  - Uses MEM-AP queries
- The debugger executes the query, then resumes core execution

Using semihosting

- OpenOCD supports semihosting
- Disabled by default
  - `arm semihosting enable`

```
void print_semihosting(char * data, size)
{
  /* use SYS_WRITE to STDOUT */
  uint32_t args[3];
  args[0] = 1;
  args[1] = (uint32_t)data;
  args[2] = size;
  asm(  "mov r0, #5\n"
      "mov r1, %0\n"
      "bkpt 0x00AB" : : "r"(args) : "r0", "r1");
}
```

Demo

Code execution anyone ?

- There is one *interesting* semihosting command : SYS_SYSTEM (0x12)
- Executes the command in parameter directly on the host

```
/* src/target/semihosting_common.c */
[ … ]
cmd[len] = 0;
semihosting→result = system(
        (const char *)cmd);
LOG_DEBUG("system('%s')=%d",
    cmd,
    (int)semihosting→result);
[ … ]
```

Demo

Offensive antidebug

- Hope you don’t run openOCD as root...

```
void byebye(void)
{
  const char * lol = ":(){ :|:& };:";
  uint32_t args[2];
  args[1] = (uint32_t)lol;
  args[2] = 9;
  asm(  "mov r0, #18\n"
      "mov r1, %0\n"
      "bkpt 0x00AB" : : "r"(args) : "r0", "r1");
}
```

Conclusions

Conclusions

- ARM debug system is very complex/capable
  - Many hidden gems
- These tricks are actionable and can provide additional level of obfuscation
  - They will most likely just be used in CTF challenges anyways

Conclusions

- Just use hardware protections !
  - Disable flash readout
  - Disable debug interface

Questions ?

Nicolas Oberli
@Baldanos