Everything you ever wanted to know about debug interfaces

Talk @ Fri3d Camp 2022

PoroCYon

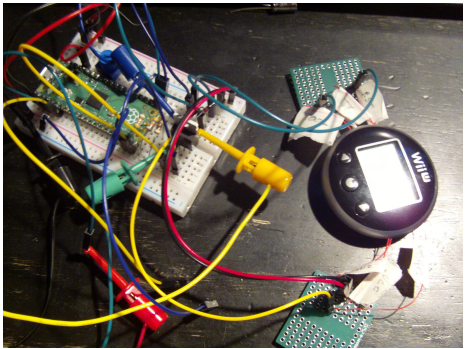
Slides: `https://pcy.be/fc22`

TOC

Whoami

- Demoscener and hardware hacker
- Dumped DSi ARM7 boot ROM and Wii Fit U Meter flash using glitching
- Linux demoscene 4k intro tooling, ...

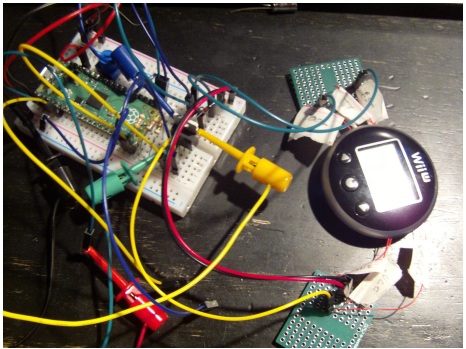
Wii Fit U Meter



- Similar to Pokéwalker, but different MCU
- No IR exploit known ($\leftrightarrow$ Pokéwalker)
- VFI attack inspired by fail0verflow on the PS4 Syscon[2]

[2] https://fail0verflow.com/blog/2018/ps4-syscon/

Wii Fit U Meter

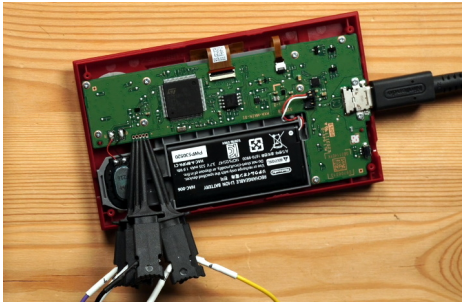


- Similar to Pokéwalker, but different MCU
- No IR exploit known ($\leftrightarrow$ Pokéwalker)
- VFI attack inspired by fail0verflow on the PS4 Syscon[2]
- Targetting the debug interface!

[2] https://fail0verflow.com/blog/2018/ps4-syscon/

Debug interfaces

- Read/write memory, single-step, breakpoints, ... on hardware
- Something controlling the CPU, but not your code??
- Security implications!



source:

https://media.ccc.de/v/rc3-11527-hacking_the_nintendo_game_watch

Debug interfaces

- Can teach you how a microcontroller *actually* works
- Often poorly documented, sadly

1	(N/A)	Always write 0, same as previous families.
2	(N/A)	Always write 0, same as previous families.
3	WAIT	Wait signal to the CPU. Read only. 1 = CPU clock stopped - waiting for an operation to 0 = CPU clock not stopped
4	BYTE	Controls the BYTE signal of the CPU used for mem 1 = Byte (8-bit) access 0 = Word (16-bit) access

12	RELEASE_LBYTE0	Release control bits in low byte from JTAG control.
13	RELEASE_LBYTE1	00 = All bits are controlled by JTAG if TCE1 is 1 01 = RW (bit 0) and BYTE (bit 4) are released from 10 = RW (bit 0), HALT (bit 1), INTREQ (bit 2), and E 11 = Reserved
14	INSTR_SEQ_NO0	Instruction sequence number. Read only.

Debug interfaces

- Creating a CPU: “black magic”
- Creating a debugger: “extreme ultra evil elite black magic”

I assume you:

- know what a debugger is (or have used one)
- have an idea about what an assembly instruction is
- have a vague idea what Arudino is
- have a very vague concept of digital hardware (eg. have played Turing Complete)

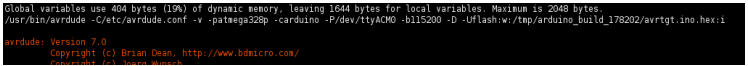
What does this button do?

avrdude

```
Global variables use 404 bytes (19%) of dynamic memory, leaving 1644 bytes for local variables. Maximum is 2048 bytes.
/usr/bin/avrdude -C/etc/avrdude.conf -v -patmega328p -carduino -P/dev/ttyACM0 -b115200 -D -Uflash:w:/tmp/arduino_build_178202/avrtgt.ino.hex:i

avrdude: Version 7.0
         Copyright (c) Brian Dean, http://www.bdmicro.com/
         Copyright (c) Joerg Wunsch
```

```
         Programmer Type : Arduino
         Description     : Arduino
         Hardware Version: 3
         Firmware Version: 4.4

avrdude: AVR device initialized and ready to a

Reading | ##################################

avrdude: Device signature = 0x1e950f (probably
avrdude: reading input file "/tmp/arduino_buil
avrdude: writing flash (3304 bytes):

Writing | ##################################
```

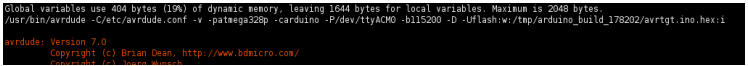
avrdude



- `avrdude`: does heavy lifting of the actual upload
- From the `avrdude` documentation:

  The Arduino [...] is supported via its own programmer type specification "arduino". This programmer works for the Arduino Uno Rev3 or any AVR that runs the Optiboot bootloader.

avrdude



- `avrdude`: does heavy lifting of the actual upload
- From the `avrdude` documentation:

  The Arduino [...] is supported via its own programmer type specification "arduino". This programmer works for the Arduino Uno Rev3 or any AVR that runs the Optiboot bootloader.

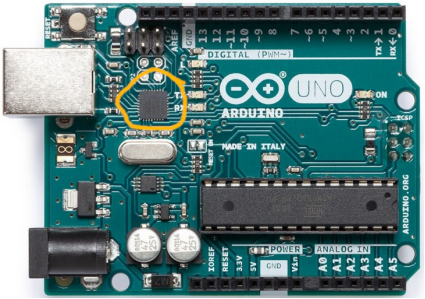
- Optiboot??

Optiboot

- 'Bootloader': small program in flash that loads your Arduino sketch
- But the ATmega328P cannot speak USB directly

Optiboot

- 'Bootloader': small program in flash that loads your Arduino sketch
- But the ATmega328P cannot speak USB directly
- ⇒ Secret second ATmega on the Arduino!



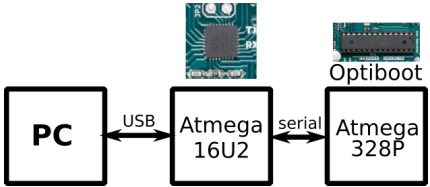


image source: https://corzotech.com/en/boards/202-arduino-uno-rev3-with-long-pins-8058333491769.html
https://github.com/arduino/ArduinoCore-avr/tree/master/bootloaders/optiboot

https://github.com/arduino/ArduinoCore-avr/tree/master/firmwares/atmegaxxu2

Is this an answer

No!

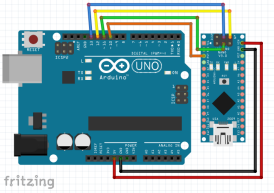
How does the bootloader get inside the ATmega328P?
How does the USB↔serial firmware get inside the ATmega16U2?

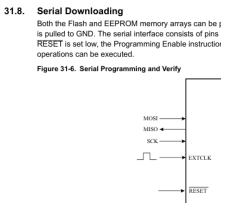
ICSP

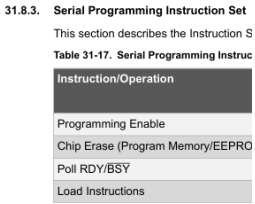
In **C**ircuit **S**erial **P**rogramming

- SPI-based protocol
- Device identification
- Read & write memory
- No debug capability

Arduino-ISP!







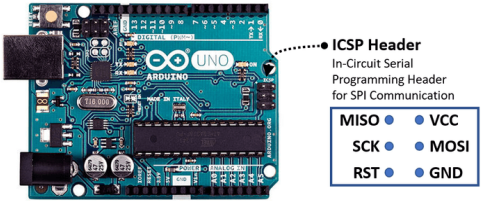


image sources: ATmega328P datasheet 42735,
https://arduinoaddiction.blogspot.com/2016/02/program-arduino-nano-via-uno-with-icsp.html,
https://www.e-tinkers.com/2020/03/do-you-know-arduino-spi-and-arduino-spi-library/

Debug?

debugWire

- Single-wire UART
- Simple command set: command X = do XYZ
- Access to CPU registers & breakpoints
- No official docs, but reverse-engineered

http://www.ruemohr.org/docs/debugwire.html

Resuming execution

```
D0 00 00 xx -- set PC, xx = 40/60 - 41/61 - 5
D1 00 01 -- set breakpoint (single step in th
D0 00 00 30 -- set PC and GO
```

Resuming from a SW BP

```
D0 00 00 79/59 -- set PC
D1 00 01 -- set breakpoint (single step in th
D2 ii ii -- load the instruction replaced by
D0 00 00 32 -- set PC and GO
```

Step Out -- D1 isn't used

```
D0 00 00 63/43 -- set PC
D0 00 00 30 -- set PC and GO
```

source: see URL

Annoying limitation

- dW $\leftrightarrow$ ICSP: can only use one
- Annoying mode switches between the two

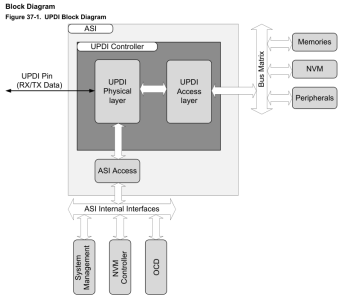
- New protocols: **T**iny **P**rogramming **I**nterface + (**U**nified) **P**rogram and **D**ebug **I**nterface
- Used in new AVRs (tinyAVR, megaAVR, AVR-Dx)

source: Hackaday

New debug protocols

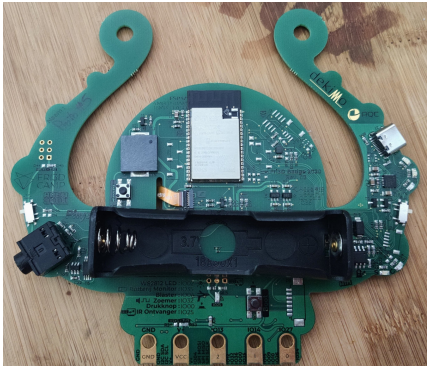
- `TPI`: in *tinyAVR*: clocked serial
- `PDI`: in *megaAVR*: uses `JTAG`
- `UPDI`: in *AVR-Dx* and *tinyAVR*: single-wire `UART`

- Use same principle: control complex state machine, send instructions to access memory (program + data) and debug stuff
- `KEY` enables features (flashing, debugging, ...)
- Debug parts undocumented (but partly reverse-engineered) `https://dragonmux.github.io/tempest/`

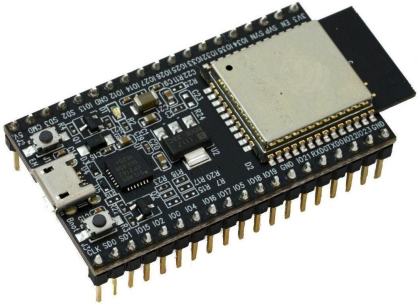


source: DS40002311A ATtiny424 datasheet

Next target

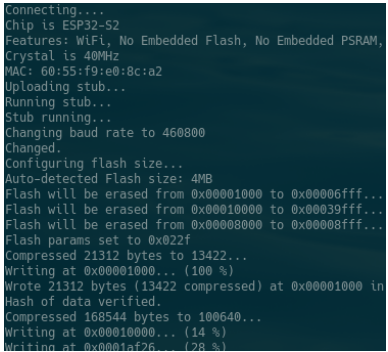


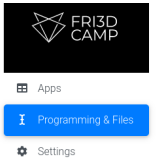
source: https://github.com/Fri3dCamp/badge-2020

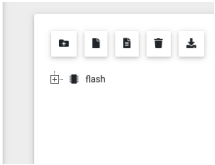


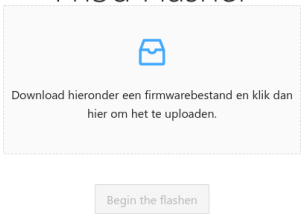
source: https://www.nabto.com/guide-to-iot-esp-32/

ESP32 programming









A bootrom!

First stage bootloader

After SoC reset, PRO CPU will start running immediately, executing reset vector code, while APP CPU will be held in reset. During startup process, PRO CPU does all the initialization. APP CPU reset is de-asserted in the `call_start_cpu0` function of application startup code. Reset vector code is located in the mask ROM of the ESP32 chip and cannot be modified.

2. For power-on reset, software SOC reset, and watchdog SOC reset: check the `GPIO_STRAP_REG` register if a custom boot mode (such as UART Download Mode) is requested. If this is the case, this custom loader mode is executed from ROM. Otherwise, proceed with boot as if it was due to software CPU reset. Consult ESP32 datasheet for a description of SoC boot modes and how to execute them.

source: `https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/startup.html`

- Bootrom: immutable program running in CPU
- Uses UART to load program & save to flash
- $\leftrightarrow$ bootloader: this not in flash!

A bootrom!

First stage bootloader

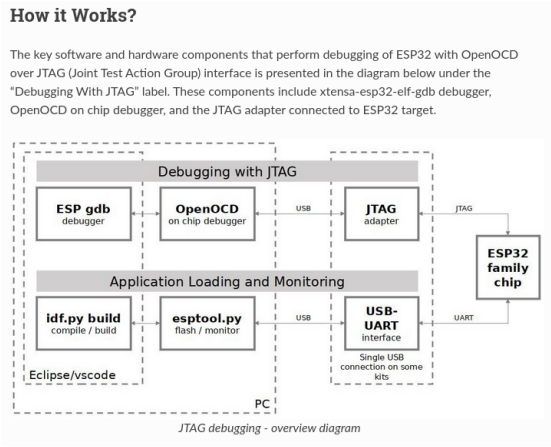
After SoC reset, PRO CPU will start running immediately, executing reset vector code, while APP CPU will be held in reset. During startup process, PRO CPU does all the initialization. APP CPU reset is de-asserted in the `call_start_cpu0` function of application startup code. Reset vector code is located in the mask ROM of the ESP32 chip and cannot be modified.

2. For power-on reset, software SOC reset, and watchdog SOC reset: check the `GPIO_STRAP_REG` register if a custom boot mode (such as UART Download Mode) is requested. If this is the case, this custom loader mode is executed from ROM. Otherwise, proceed with boot as if it was due to software CPU reset. Consult ESP32 datasheet for a description of SoC boot modes and how to execute them.

source: `https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/startup.html`

- Bootrom: immutable program running in CPU
- Uses UART to load program & save to flash
- $\leftrightarrow$ bootloader: this not in flash!
- .. can't do debug, but ESP32 has a debugger!

ESP32 debug

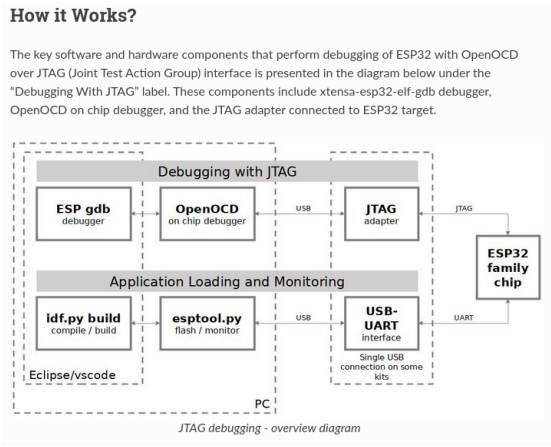


JTAG debugging - overview diagram

source: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/

- Based on OpenOCD
- Uses JTAG
- Protocol not documented

ESP32 debug

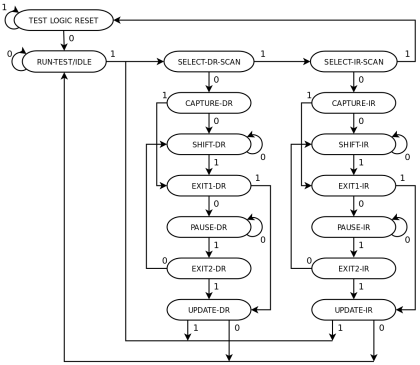


JTAG debugging - overview diagram

source: `https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/`

- Based on OpenOCD
- Uses JTAG
- Protocol not documented
- OpenOCD is FOSS...

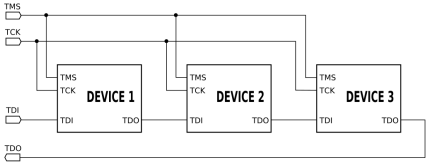
JTAG



source: `https://commons.wikimedia.org/wiki/File:JTAG_TAP_Controller_State_Diagram.svg`

- 4 wires: `TMS`, `TCK`, `TDI`, `TDO`
- Complex state machine to access two registers
- Registers give access to everything
- Some standardization, lots of legacy

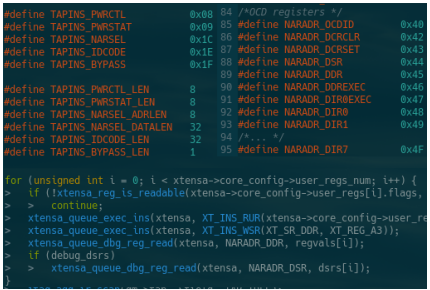
JTAG: why?



source: https://commons.wikimedia.org/wiki/File:Jtag_chain.svg

- Device identification
- "Boundary scan"
- Flexible: register sizes, instruction numbers not defined
- Now used mainly for debugging instead

JTAG on the ESP32



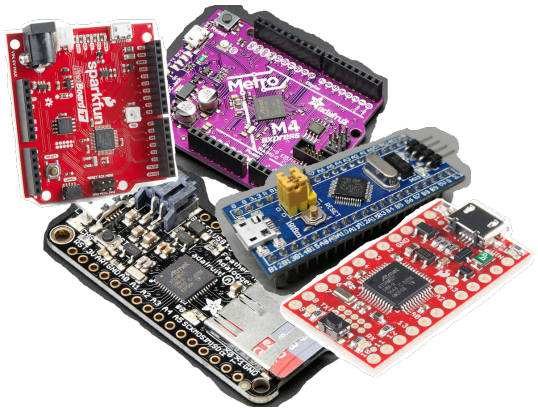
```
#define TAPINS_PWRCTL          0x08
#define TAPINS_PWRSTAT         0x09
#define TAPINS_NARSEL          0x1C
#define TAPINS_IDCODE          0x1E
#define TAPINS_BYPASS          0x1F

#define TAPINS_PWRCTL_LEN      8
#define TAPINS_PWRSTAT_LEN     8
#define TAPINS_NARSEL_ADRLEN   8
#define TAPINS_NARSEL_DATALEN  32
#define TAPINS_IDCODE_LEN      32
#define TAPINS_BYPASS_LEN      1

for (unsigned int i = 0; i < xtensa->core_config->user_regs_num; i++) {
    if (!xtensa_reg_is_readable(xtensa->core_config->user_regs[i].flags,
        continue;
    xtensa_queue_exec_ins(xtensa, XT_INS_RUR(xtensa->core_config->user_re
    xtensa_queue_exec_ins(xtensa, XT_INS_WSR(XT_SR_DDR, XT_REG_A3));
    xtensa_queue_dbg_reg_read(xtensa, NARADR_DDR, regvals[i]);
    if (debug_dsrs)
        xtensa_queue_dbg_reg_read(xtensa, NARADR_DSR, dsrs[i]);
}
```

```
/*OCD registers */
#define NARADR_OCDID        0x40
#define NARADR_DCRCLR       0x42
#define NARADR_DCRSET       0x43
#define NARADR_DSR          0x44
#define NARADR_DDR          0x45
#define NARADR_DDREXEC      0x46
#define NARADR_DIR0EXEC     0x47
#define NARADR_DIR0         0x48
#define NARADR_DIR1         0x49
/*... */
#define NARADR_DIR7         0x4F
```

- No docs, but OpenOCD source code
- Xtensa ‘NAR’, ‘TRAX’, and CoreSight CTI
- ‘NAR’: send single instructions to CPU, use `DDR` as data channel

`https://github.com/espressif/openocd-esp32`

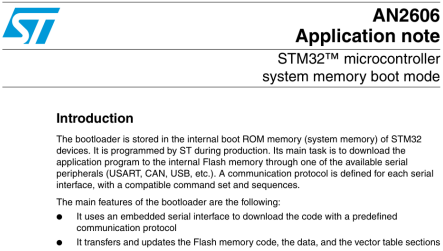
Ever seen these?



ATSAMD, STM32, LPCxx, K32L, RP2040, ...

sources: Adafruit and Sparkfun product catalogs, https://stm32-base.org/boards/STM32F103C8T6-Blue-Pill.html

Bootroms



AN2606
Application note

STM32™ microcontroller system memory boot mode

Introduction

The bootloader is stored in the internal boot ROM memory (system memory) of STM32 devices. It is programmed by ST during production. Its main task is to download the application program to the internal Flash memory through one of the available serial peripherals (USART, CAN, USB, etc.). A communication protocol is defined for each serial interface, with a compatible command set and sequences.

The main features of the bootloader are the following:

- It uses an embedded serial interface to download the code with a predefined communication protocol
- It transfers and updates the Flash memory code, the data, and the vector table sections

- Many of these have a UART bootrom
- Similar features: read & write flash, access lock, ...
- *Bootrom controls debug enable/disable* $\Rightarrow$ glitch target

ARM ADI

- ARM debug is standardized and documented!
- Many layers: `SWD` - `DP` - `MEM-AP`

SWD and the Debug Port

- Half-duplex synchronous serial
- Few registers that give access to *access port*
- Simple! ($\leftrightarrow$ `JTAG`)
- Access to MCU memory
- Extra secret registers for device info & debug control

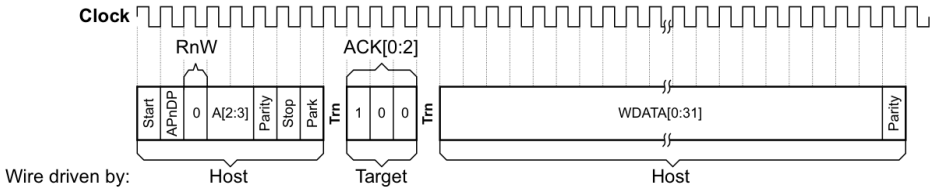


image source: ARM Debug Interface Architecture Specification v5

SWD and the Debug Port

- Half-duplex synchronous serial
- Few registers that give access to *access port*
- Simple! ($\leftrightarrow$ `JTAG`)
- Access to MCU memory
- Extra secret registers for device info & debug control

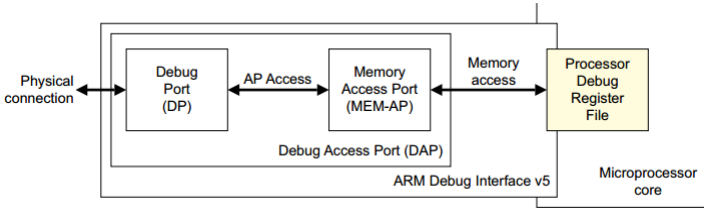


image source: ARM Debug Interface Architecture Specification v5

MEM-AP

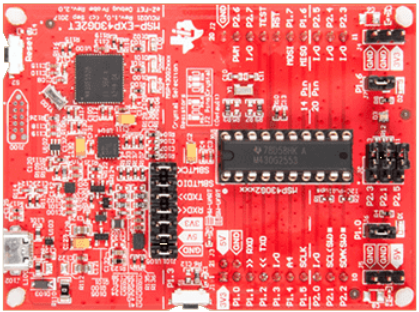
Debug resource	Address Range
Data Watchpoint and Trace	0xE0001000-0xE0001FFF
Breakpoint unit	0xE0002000-0xE0002FFF
SCS	0xE000ED00-0xE000EEFF
System Control Block	0xE000ED00-0xE000ED8F
Debug Control Block	0xE000EDF0-0xE000EEFF
ARMv6-M ROM table	0xE00FF000-0xE00FFFFF

source: ARMv6-M Architecture Reference Manual

- Accessed through SWD-DP
- Access to MCU memory
- ROM tables
- Extra registers for debug

PDP-11 in a microcontroller

source: https://energia.nu/



source: https://www.ti.com/tool/MSP-EXP430G2ET

Another bootrom

User's Guide

MSP430™ Flash Devices Bootloader (BSL)

TEXAS INSTRUMENTS

ABSTRACT

The MSP430™ bootloader (BSL) (formerly known as the bootstrap loader) allows users to communicate with embedded memory in the MSP430 microcontroller (MCU) during the prototyping phase, final production, and in service. Both the programmable memory (flash memory) and the data memory (RAM) can be modified as required. Do not confuse the bootloader with the bootstrap loader programs found in some digital signal processors (DSPs) that automatically load program code (and data) from external memory to the internal memory of the DSP.

To use the bootloader, a specific BSL entry sequence must be applied. An added sequence of commands initiates the desired function. A bootloading session can be exited by continuing operation at a defined user program address or by the reset condition.

If the device is secured by disabling JTAG, it is still possible to use the BSL. Access to the MSP430 MCU memory through the BSL is protected against misuse by the BSL password. The BSL password is equal to the content of the interrupt vector table on the device.

source: `https://www.ti.com/lit/pdf/slau319`

- UART/I^2C
- Read from flash is password-protected
- Erase flash on wrong password!

Another bootrom

User's Guide

MSP430™ Flash Devices Bootloader (BSL)

TEXAS INSTRUMENTS

ABSTRACT

The MSP430™ bootloader (BSL) (formerly known as the bootstrap loader) allows users to communicate with embedded memory in the MSP430 microcontroller (MCU) during the prototyping phase, final production, and in service. Both the programmable memory (flash memory) and the data memory (RAM) can be modified as required. Do not confuse the bootloader with the bootstrap loader programs found in some digital signal processors (DSPs) that automatically load program code (and data) from external memory to the internal memory of the DSP.

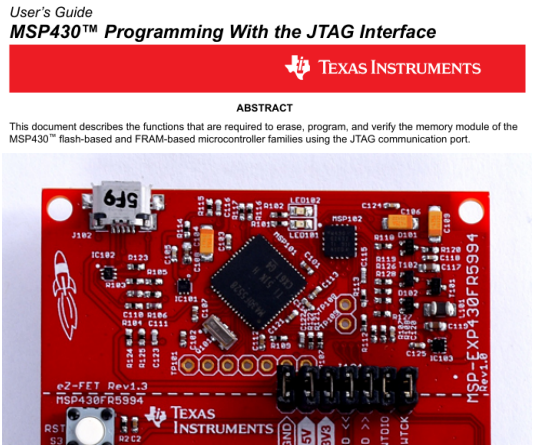
To use the bootloader, a specific BSL entry sequence must be applied. An added sequence of commands initiates the desired function. A bootloading session can be exited by continuing operation at a defined user program address or by the reset condition.

If the device is secured by disabling JTAG, it is still possible to use the BSL. Access to the MSP430 MCU memory through the BSL is protected against misuse by the BSL password. The BSL password is equal to the content of the interrupt vector table on the device.

source: `https://www.ti.com/lit/pdf/slau319`

- UART/I^2C
- Read from flash is password-protected
- Erase flash on wrong password!
- "authenticated" flag stored in RAM...
- Target 'authenticated' check for RAM write command

Debugger woes


User's Guide

MSP430™ Programming With the JTAG Interface

TEXAS INSTRUMENTS

ABSTRACT

This document describes the functions that are required to erase, program, and verify the memory module of the MSP430™ flash-based and FRAM-based microcontroller families using the JTAG communication port.

- Every devboard has a debugger
- Documented in a PDF
- ... kinda

sources: https://www.ti.com/lit/pdf/slau320,

Physical layer

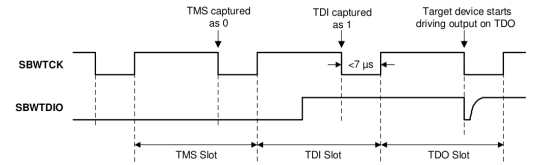
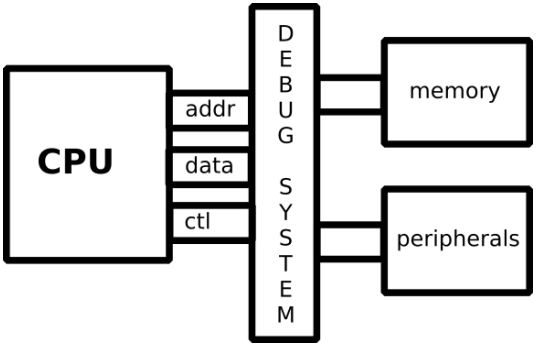


Figure 2-8. Spy-Bi-Wire Timing Diagram

source: `https://www.ti.com/lit/pdf/slau320`

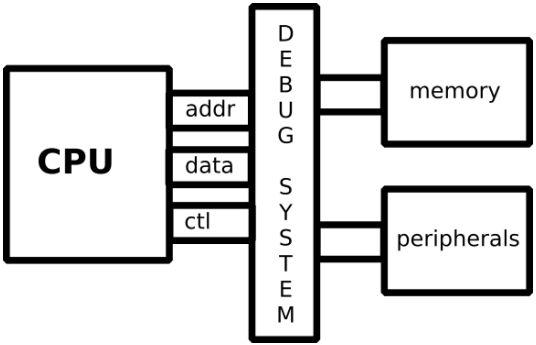
- Spy-Bi-Wire: 2-wire JTAG
- Time multiplexed
- Extra stuff: `TCLK`: send clock edges to CPU

Debug layer



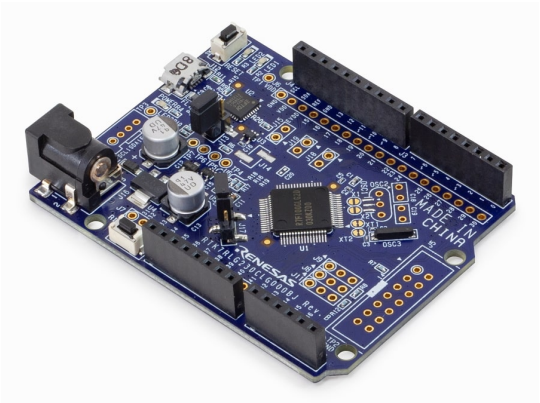
- CPU is connected to outside world and memory using the *system bus*
- Debug system: sit between CPU and bus
- Control addresses, data, and CPU signals

Debug layer



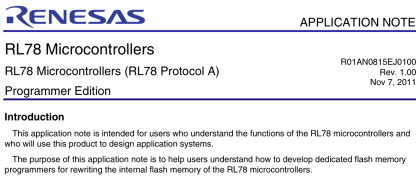
- CPU is connected to outside world and memory using the *system bus*
- Debug system: sit between CPU and bus
- Control addresses, data, and CPU signals
- Very low-level control: detailed but hard to use

Z80, but worse



source: https://www.renesas.com/sites/default/files/rl78-g23-64p-fpb-board.jpg

Bootrom time



RENESAS APPLICATION NOTE

RL78 Microcontrollers

RL78 Microcontrollers (RL78 Protocol A)
Programmer Edition

R01AN0815EJ0100
Rev. 1.00
Nov 7, 2011

Introduction

This application note is intended for users who understand the functions of the RL78 microcontrollers and who will use this product to design application systems.

The purpose of this application note is to help users understand how to develop dedicated flash memory programmers for rewriting the internal flash memory of the RL78 microcontrollers.

- UART bootrom
- Documented in PDF
- No flash read command $\Rightarrow$ need something better

source: `https://www.renesas.com/eu/en/document/apn/rl78-microcontrollers-rl78-protocol-programmer-edition-application-note-rev100`

Reversing the bootrom

Table 1-5. 1-byte Data and Communication Interface of RL78

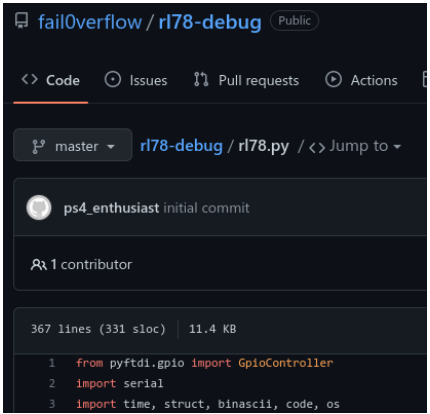
1-byte Data	Communication Interface
3AH	Single-wire UART
00H	Two-wire UART

```
  if (tool0_in == 0x3a) {
set_tool_en:
    write_volatile_1(REG_TOOLEN,uVar6);
    tool0_in = read_volatile_1(REG_BACDMH0);
    write_volatile_1(REG_BACDMH0,tool0_in | 0x10);
  }
  else {
    uVar6 = 0x35;
    bVar18 = false;
    if (tool0_in == 0) goto set_tool_en;
    bVar18 = tool0_in < 0xc5;
    do {
    } while (tool0_in != 0xc5);
    use_proto_ocd_flag = use_proto_ocd_flag | 0x20;
  }
```

source: https://www.renesas.com/eu/en/document/apn/rl78-microcontrollers-rl78-protocol-programmer-edition-application-note-rev100

- Docs specify entry modes
- Bootrom uses an extra one
- ... it enables a debugger command set

Debugger



source:

https://github.com/fail0verflow/rl78-debug/blob/master/rl78.py

- Command set: read & write memory, execute asm snippet
- Implemented in bootrom, no special hardware!
- Has “erase on wrong password” feature, but.. not implemented
- Very hackable and glitchable (workshop Sunday morning: learn how to hack it yourself!)

Only the beginning

- Skipped eg. ARM CTI, MSP430 EEM, instruction tracing, EnergyTrace, ...
- More detail in protection mechanisms
- Other protocols
  - RISC-V
  - Nexus (OpenMSP430, AVR32, MPC/SPC, ...)
  - PIC
  - STM8 SWIM
  - MAXQ JTAG
  - EFM8 C2
  - ...
- History (eg. old ARM EmbeddedICE)
- Core↔core debug (Nailgun hack![1])
- How to actually implement this

[1] *Understanding the Security of ARM Debugging Features*, Ning & Zhang

Conclusion

- Debug systems are interesting
- Many different ways of making one

- Attrative target for hacking
- Need to understand how they work to know the risks
- But companies won't tell you enough to know the risks

Questions

Questions?

Fedi/masto: `@pcy@icosahedron.website`
Mail me at `p@pcy.be`
`pcy` on Fri3d Camp Discord
Slides available at `https://pcy.be/fc22`