arm

pyOCD introduction for TF-M

v1.0

Chris Reed

Oct 2021

Agenda

- Introduction
- Features and roadmap
- Getting started
  - Probes and targets
  - Installing target support
  - Configuration
  - Programming memory
- Debugging: gdb and VSCode
- Q&A

Introduction

pyOCD == **Py**thon **O**n **C**hip **D**ebugger
https://pyocd.io/

Open source: https://github.com/pyocd/pyOCD

Apache 2.0 license

Distributed as a Python package via PyPI

- Install via pip/pipx

- General debug
- CI and test
- Manufacturing, provisioning
- Bespoke debug scripts, tools, utilities
- Security research
- SoC and board bring-up

Originally created by the Mbed team within Arm

Now an independent project

Why pyOCD?

What makes it different and worth using? (Especially compared to OpenOCD.)

Key distinctions

1. Best for Arm
   - Integrates with Arm ecosystem and CMSIS.
2. Focus on ease of use
   - But still retaining configurability and extensibility.
3. Python
   - Easy to integrate for CI, test, bespoke debug tools, etc.
4. Permissive open source license (Apache 2.0)

Major features

- CMSIS Device Family Pack support
- Standard CMSIS flash algo support
- CoreSight discovery
  - No hard-coded config (generally)
- Easy to use Python API
- RTOS awareness
- SWO/SWV
- ADIv6 support (e.g., Cortex-M55)
- TCP debug probe server/client
- SVD register access via commands
- Plug-ins

Roadmap

Where pyOCD is headed next.

Short term/in progress:

- CMSIS DFP debug sequences
- Better TrustZone-M support (work around gdb)
- Reusable debug controller class
- Event Recorder, aka CMSIS View
- Segger RTT
- Built-in debug authentication (via SDM API and PSA ADAC)

Longer term:

- Microsoft Debug Adapter Protocol
- Cortex-A
- IO expansion (I2C, SPI, GPIO)
- Trace via ETB/MTB and TPIU
- Board-level config (QSPI algos, etc.)
- Support for Fast Models
- More extensibility
- …
- Long term goal: Full debug capability?

Getting started

pyOCD command line tool

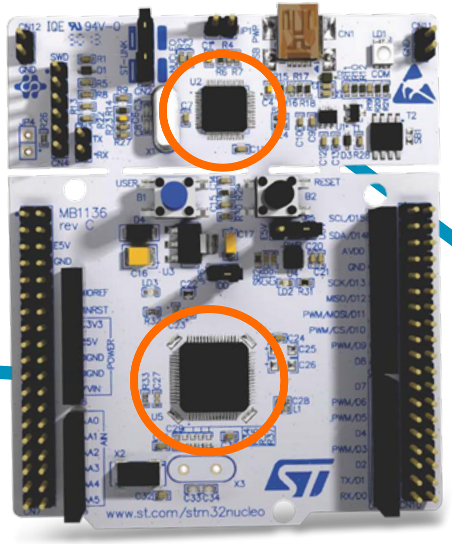
The primary interface to pyOCD is through the `pyocd` command line tool with these subcommands:

Subcommand	Description	Connects?
`list`	Display available debug probes, targets, boards, plugins.	N
`gdbserver, gdb`	Start gdbserver for debugging.	Y
`pack`	Manage CMSIS Device Family Packs that provide target support.	N
`load, flash`	Program files into memory, RAM and flash.	Y
`erase`	Erase chip or range of sectors.	Y
`commander, cmd`	REPL for interactively investigating devices.	Y
`json`	Similar to *`list`* but JSON output.	N
`server`	Serve debug probe via TCP/IP.	Y

pyOCD needs to know…

Subcommands that control the MCU have a set of common arguments.

On-board debug probe



1. What MCU to debug?
⟹ **Target**

2. How to talk to it?
⟹ **Debug probe**

(Implicitly, *which* MCU to debug.)

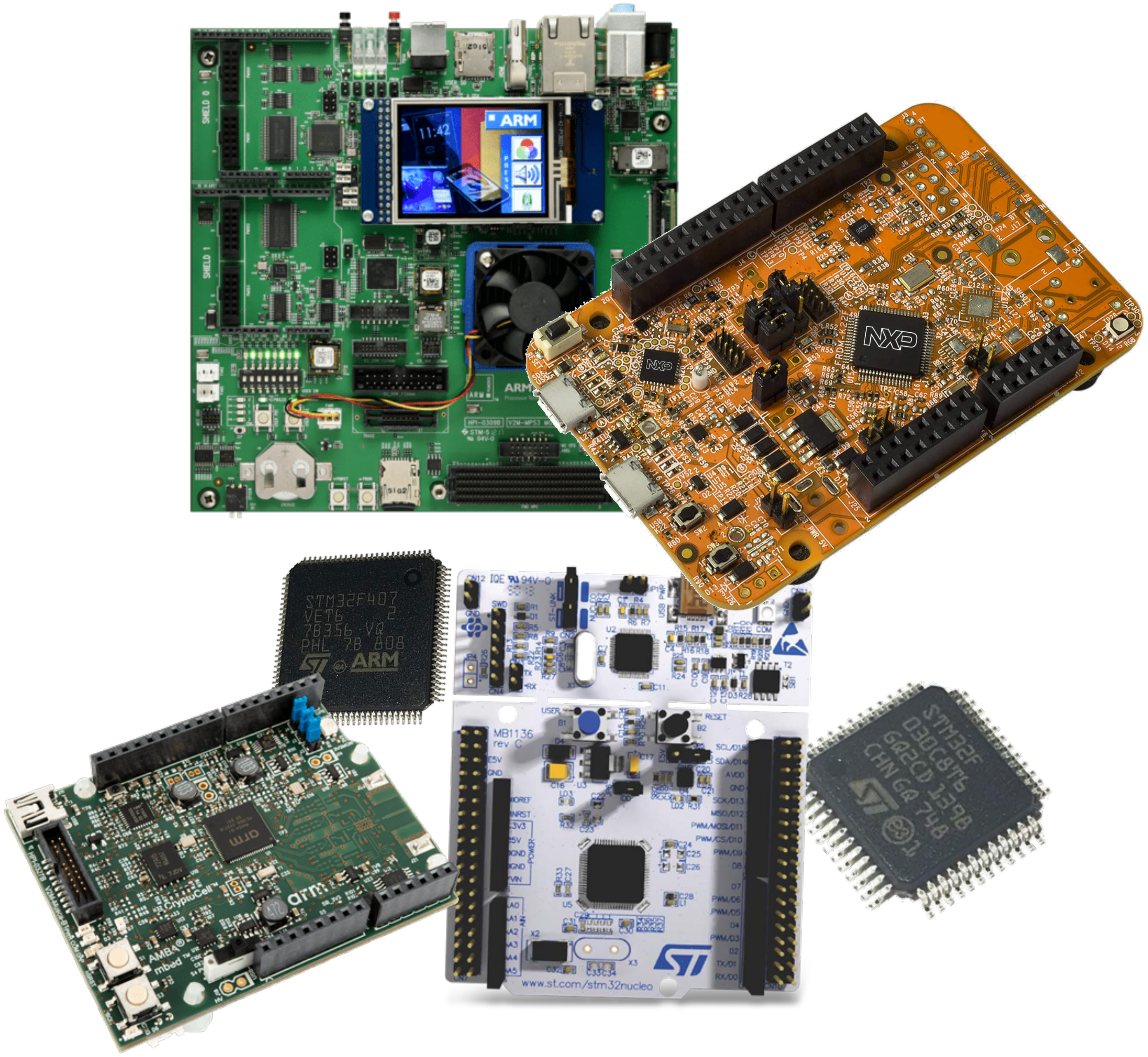
Standalone debug probe

Targets

- **Target**: the MCU being debugged
- **Target type**: the MCU family and part number

- Target types combine:
  - Memory map
  - Flash programming algorithms
  - Special debug logic
  - Other info

- 70+ built-in target types
- Most other Cortex-M devices supported via CMSIS Device Family Packs



Debug probes

- The interface that drives SWD or JTAG to the MCU
- Two flavours:
  - **On-board probes**
    - Ex: DAPLink on Arm Musca boards
    - Ex: STLink on STMicro Nucleo boards
  - **Standalone probes**
    - Ex: Arm ULINKplus
    - Ex: Segger J-Link
- Supported probe types:
  - CMSIS-DAP v1 (HID) and v2 (WinUSB)
  - STLinkV2/V3
  - J-Link
  - Raspberry Pi RP2040 picoprobe
  - PE Micro
  - TCP/IP remote probe server (pyocd proprietary protocol)

These are all standalone debug probes…

Selecting the debug probe

- Every debug probe has a **unique ID**.
  - View by running `pyocd list`.

```
#   Probe                                     Unique ID
-------------------------------------------------------------------------------------------------------
0   Arm LPC55xx DAPLink CMSIS-DAP             000000803f7099a85fdf51158d5dfcaa6102ef474c504355
1   Arm Musca-B1 [musca_b1]                   500700001c16fcd4000000000000000000000000097969902
2   Arm V2M-MPS3 [cortex_m]                   5005000019150e4b000000000000000000000000097969902
3   DISCO-H747I [stm32h747xihx]               001700343137510D39383538
4   FRDM-K64F [k64f]                          02400b0129164e4500440012706e0007f301000097969900
5   MIMXRT1050-EVKB [mimxrt1050_hyperflash]   02270b0341114e450014300ac207002392d1000097969900
6   NUCLEO-H743ZI2 [stm32h743zitx]            002100075553500E20393256
7   Segger J-Link OB-K22-NordicSemi           960177309
```

- Three methods to select the probe:

1. Only one probe is connected: pyOCD selects it automatically.
2. Multiple probes are connected: pyOCD asks you to select a probe before continuing.
3. Explicitly select with `-u UID` / `--uid=UID` / `--probe=UID`
   - Can restrict probe type with `plugin-name:` prefix on UID.

Specifying the target type

- Each target type has a name
  - e.g., “k64f”, “stm32l475xg”, “nrf5340_xxaa”, “k32l3a60vpj1a”
  - Often the full part number, except built-in targets tend to have short names

- Many on-board debug probes know their connected target type.
  - DAPLink firmware and STLinkV2/V3 support this
- For standalone probes you *must* tell pyOCD.

- Set with `-t TARGET` / `--target=TARGET`
  - Or with a config file

- Default target type is “cortex_m”
  - Architectural memory map
  - No flash programming
  - No custom target debug logic
  - ⟶ pyocd warns if cortex_m gets used by default.

Probes’ connected target type is in brackets if known.

```
  #   Probe                                      Unique ID
-----------------------------------------------------------------------------------------------
  0   Arm LPC55xx DAPLink CMSIS-DAP              000000803f7099a85fdf51158d5dfcaa6102ef474c504355
  1   Arm Musca-B1 [musca_b1]                    500700001c16fcd400000000000000000000000097969902
  2   Arm V2M-MPS3 [cortex_m]                    5005000019150e4b00000000000000000000000097969902
  3   DISCO-H747I [stm32h747xihx]                001700343137510D39383538
  4   FRDM-K64F [k64f]                           02400b0129164e4500440012706e0007f301000097969900
  5   MIMXRT1050-EVKB [mimxrt1050_hyperflash]    02270b0341114e450014300ac207002392d1000097969900
  6   NUCLEO-H743ZI2 [stm32h743zitx]             002100075553500E20393256
  7   Segger J-Link OB-K22-NordicSemi            960177309
```

Checking and installing target support

- Two sources of target support:
    1. Built-in
    2. CMSIS Device Family Packs (DFPs)

- Check for target type with `pyocd list --targets --name TARGET-TYPE-NAME`
    - Will print all matching installed targets and the source.
    - Partial target type names are accepted; match is case-independent.
    - Be aware that built-in target type names are usually not the full part number.

- To find and install CMSIS DFP target support:
    - `pyocd pack find PART-NUMBER`
    - `pyocd pack install PART-NUMBER`
    - Partial names are accepted; match is case-independent.

Configuration

- "Session options" can be set in several ways:
  - Many common session options have dedicated command line arguments.
  - Passed on pyocd command line with `-Ooption[=value]` arguments.
  - Place in a `pyocd.yaml` config file in your project directory.
- Config files support both global and probe-specific options.
  - Probe-specific config is very useful for setting the target type of standalone probes!
- Example config file:

```yaml
# Probe-specific options.
probes:
  066EFF555051897267233656: # Probe's unique ID.
    target_override: stm32l475xg

# Global options
auto_unlock: false
frequency: 8000000 # Set 8 MHz SWD default for all probes
persist: true # Make gdbserver persist after gdb disconnects
```

- Session option documentation: https://pyocd.io/docs/options.html

Programming memory

Usage: `pyocd load <file> [<file>…]`

- Use to quickly program one or more files to device memory (flash and/or RAM).
- Accepts binary, Intel hex, and ELF files.

- To force chip or sector erase:

  `--erase {auto,chip,sector}`

  The default is *sector*. *auto* uses chip or sector depending on which is estimated to be fastest.

- To set the base address for binary files:
  - Append `@ADDRESS` to the binary file's name on the command line.
    - e.g., `pyocd load mybinary.bin@0x8000`
  - Or use the `-a` / `--base-address ADDRESS` argument (works only if one supplied binary file).
  - The default is to use a base address of the start of flash.
    - Again, works only for one supplied binary file.

Debugging with VSCode and Cortex-Debug

Debugging options

There are several options for how you debug using pyOCD. All rely on gdb.

1. Command line gdb
   - Go old school!

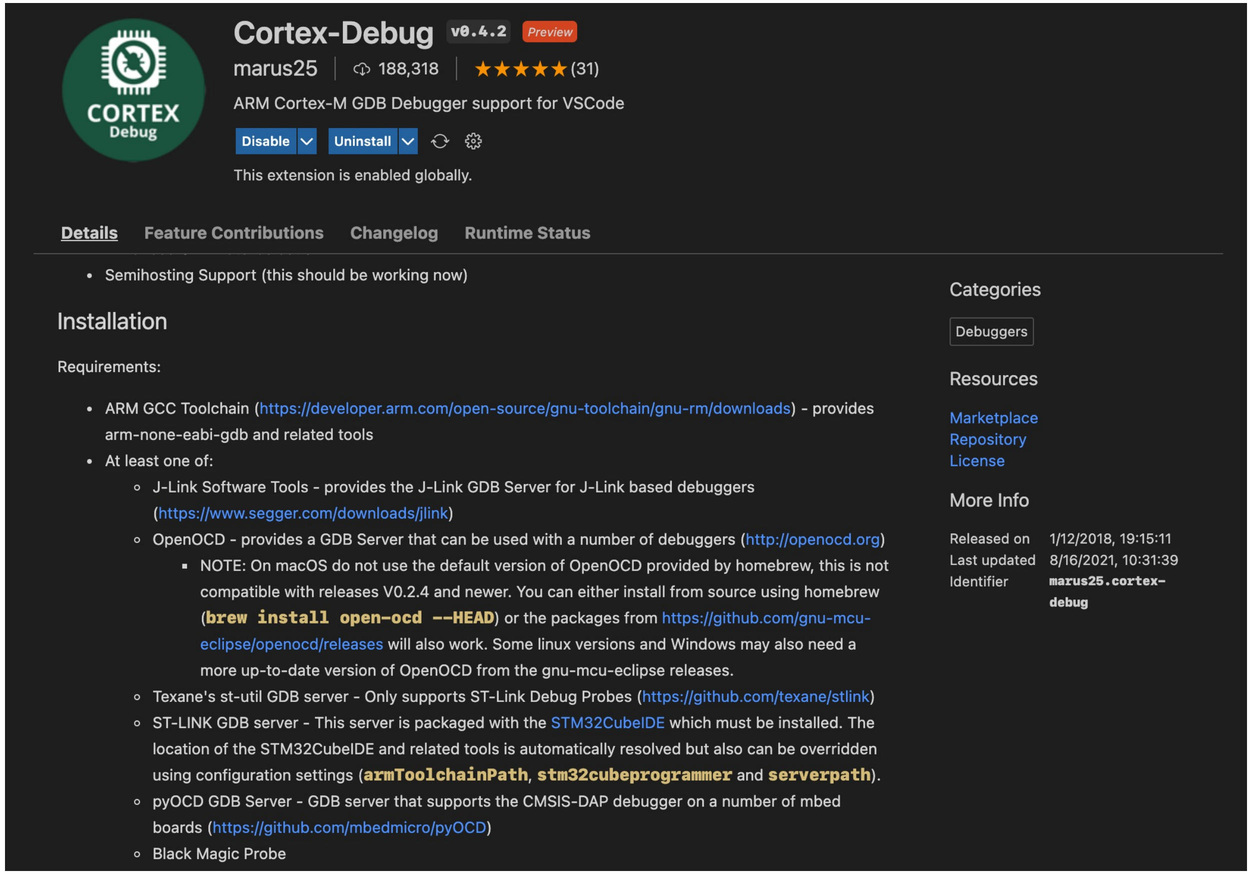
2. Visual Studio Code with Cortex-Debug extension

3. Eclipse Embedded CDT

Cortex-Debug plugin

This plugin provides a debug adaptor for arm-none-eabi-gdb with support for pyOCD and other gdbservers.

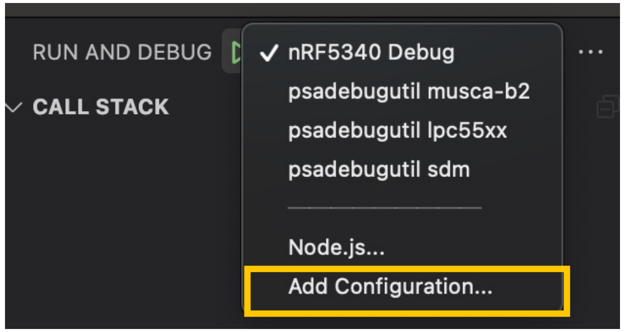
Install the [Cortex-Debug]() plugin from the extensions marketplace.



Launch configuration

- Add a configuration to `.vscode/launch.json` for your project.

- You can use the *Add Configuration…* menu item to get started.



Example `launch.json`:

```
{ "version": "0.2.0", "configurations": [
 {
  "cwd": "${workspaceRoot}",
  "executable": "${workspaceRoot}/firmware.elf",
  "name": "pyOCD Debug",
  "request": "launch",
  "type": "cortex-debug",
  "servertype": "pyocd",
  "serverpath": "<path-to-pyocd>",
  "targetId": "<target-type-name>",
  "serverArgs": [ // <- cmdline args for pyocd
    "--uid=<probe-id>", // probe unique ID
    "--core=0", // run gdbserver for only this core
  ],
  "svdFile": "<path-to-svd>",
  "showDevDebugOutput": false,
 }
] }
```

Config attribute docs: https://github.com/Marus/cortex-debug/blob/master/debug_attributes.md

The docs say to use boardId for the probe unique ID, but that uses an old pyocd command line argument.

Debugging tips

1. (This may be obvious, but…) Make sure you use a Debug build!

2. SVD files can be obtained from CMSIS Packs
   - Download pack from https://www.keil.com/dd2/pack/
   - Extract as zip file and look for SVD file

3. Add `"gdbPath": "arm-none-eabi-gdb-py"` to the launch config to enable Python in GNU-RM gdb and/or (with an absolute path) specify a gdb not in your PATH.

4. Use `"--core=N"` in `"serverArgs"` to select the core to debug on multicore targets.
   - Otherwise Cortex-Debug tells pyocd to use conflicting TCP ports, and it fails to start.

5. Cortex-Debug sometimes didn't properly terminate the pyocd process when stopping.
   - If you see an "Unable to open device: open failed" error, run `pkill -f 'pyocd gdb'`.

6. gdb may report "Ignoring packet error, continuing…" when programming flash, but these seem to be harmless.

TZ-M limitations

- Primary limitations of gdb:
  - Lack of support for multiple CPU contexts
  - Doesn’t deal well with multiple symbols loaded with the same name, such as `main()`

- For practical purposes, this restricts you to debugging one world at a time.

Cortex-Debug config tips for TF-M

1. TF-M requires some additional launch config settings due to its complexity.

2. Create separate launch configs for S and NS debug.
   - Set the `"executable"` to either `tfm_s.elf` or `tfm_ns.elf`.

1. Override the standard Cortex-Debug gdb launch script to control how the TF-M code is loaded upon connect:

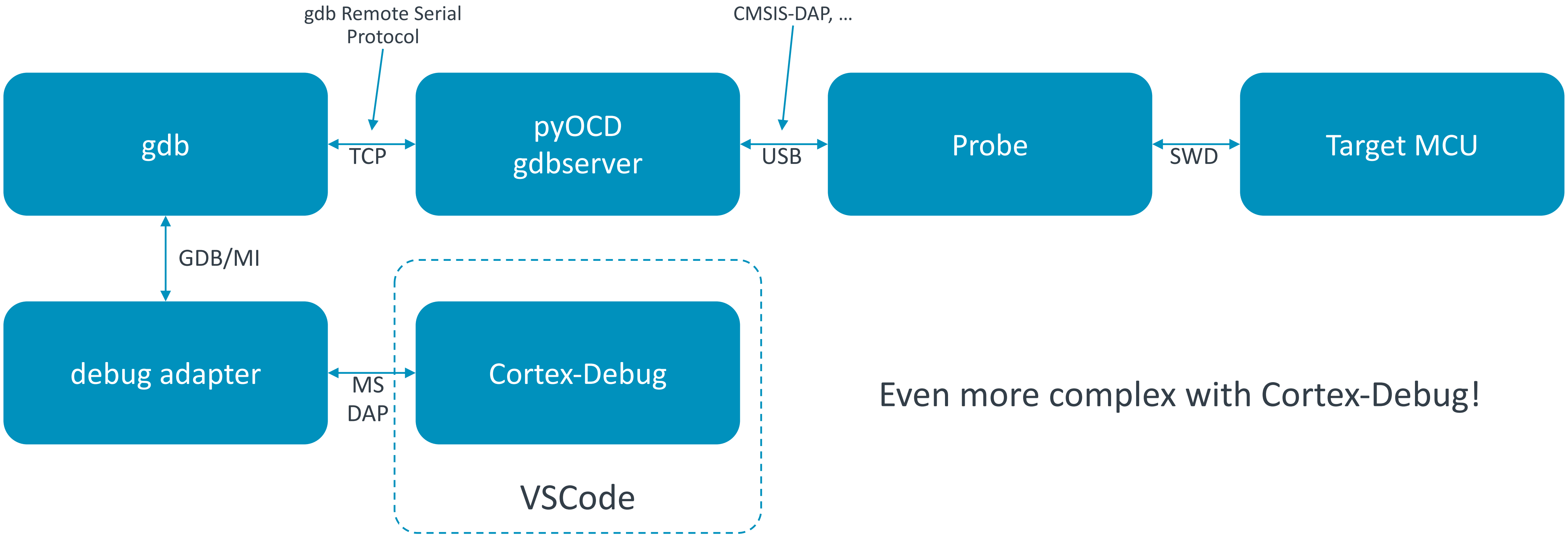
```
"overrideLaunchCommands": [
  "mon load ${workspaceRoot}/cmake_build_Debug/bin/bl2.bin 0xA000000",
  "mon load ${workspaceRoot}/cmake_build_Debug/bin/tfm_s_ns_signed.bin
0xA020000",
  "mon reset halt",
  "flushregs", // Not strictly necessary if continuing after the reset.
],
```

Q&A

Thank You
Danke
Gracias
谢谢
ありがとう
Asante
Merci
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה

Extra slides

gdbserver data flow with DAP



Even more complex with Cortex-Debug!

Flash programming options

- By default, pyOCD attempts to optimise flash programming by
  1. Choosing chip or page erase by estimating which is fastest.
  2. Not reprogramming unchanged data, including erased pages.

- These options require scanning target memory for comparison.
- When actively developing, it can boost programming speed quite a lot.
- But for large memories and situations like CI, where the new firmware is always unique, it can negatively affect performance.

Option name	Type	Default	Description
`smart_flash`	bool	true	Controls content analysis and differential programming optimisation. Set to false to use naïve programming.
`keep_unwritten`	bool	true	Whether to preserve existing flash content for ranges of sectors that will be erased but not written with new data.
`chip_erase`	str	”auto”	“auto”, “sector”, or “chip”

(Defaults may change in the future.)

erase subcommand

Usage: `pyocd erase [--chip | --sector <address-ranges…>]`

- Allows you to easily erase the entire chip or any number of sectors.
  - Only erases flash memory.

- To erase the whole chip, use the `--chip` option.
- To erase individual sectors, pass `--sector` and a list of address ranges.

Syntax	Example	Description
address	0x1000	erase single sector starting at 0x1000
start-end	0x800-0x2000	erase sectors starting at 0x800 up to but not including 0x2000
start+length	0+8192	erase 8 kB starting at address 0

The erased range will be *rounded up* to the next whole sector.