In-Circuit Debugging

When simulations suggest your program will work, and it doesn't, and a careful study of your code doesn't improve things, and well-placed `printf()` statements and LED blinks don't reveal the problem, it's time to get serious. Getting serious (i.e., using a formal debugger) is a double-edged sword because a) it can be a very powerful way to reveal program errors, but b) it can take a long time to set up, c) may require additional hardware, d) may not produce any results due to Heisenbugs, and e) may consume lots of time in debugging the setup/hardware before you ever get to debugging your own code.

Prologue

I will warn you ahead of time: it is very tempting to address all software problems by starting up the hardware debugger. My experience and opinion is that THIS IS RARELY REQUIRED! This section introduces you to the LAST LINE OF DEFENSE when debugging code, not your first option (this is, of course, just my opinion). Debugging can be very time-consuming and does not always yield results. Even worse, due to the differences between real-time execution and execution under debugger control, you can be led astray and waste your time chasing ghosts.

My advice is to follow the same "rules of effort vs. cost" in debugging as in development: the more effort expended on correctness and testing "in the lab", the less effort and cost there is "in the field". For debugging, this means that the more effort expended on correctness, testing, and simulation during the development phase, the less effort and wasted time there is in debugging.

The list below suggests testing and debugging strategies, in increasing order of cost/effort. The more time/effort you put into the first few items in the following list, the less you will have to spend on the last few items (which are expensive):

1. Write code carefully and defensively, thoroughly reading all datasheets and available documentation, always checking return codes, being proactive about providing hooks for debugging, checkpointing, and logging
2. Simulate (when possible)
3. READ your code
4. Have someone else read your code
5. Execute/debug code on actual hardware using `printf()` and LED's for debugging (or by enabling checkpoints and logging)
6. Perform in-system debugging using JTAG, a debug monitor, a logic analyzer, etc.

Hardware Debugging Support

Many modern microcontrollers are equipped with some form of hardware-assisted debugging facility. This support is necessary in response to higher levels of hardware integration. In "the old days", embedded systems were constructed from discrete components (e.g., microcontroller, RAM, ROM, A/D, UART, etc.) and the interconnecting wires between the components were available for attaching test probes. Test equipment such as oscilloscopes, logic probes, logic analyzers, and pattern generators could be used to observe and control the entire system. In a modern, integrated microcontroller, those interconnections are now all on the chip and it is impossible to, for example, notify a debugger when a particular variable in memory is overwritten, without some kind of support from the microcontroller.

In-circuit emulators (or ICE) are expensive pieces of test equipment that physically replace the target microcontroller with a special device that is supposed to exactly match the microcontroller's behavior, but is fully instrumented so that complex debugging operations (such as stopping the code when a variable in memory is overwritten) becomes possible.

The following article provides a more in-depth overview of in-circuit emulators and is worthwhile reading:

`http://www.embedded.com/showArticle.jhtml?articleID=23901694`

Fortunately, many modern microcontrollers now have built-in ICE-like peripherals that provide access to the innards of the microcontroller and enable hardware-level debugging (though not with the same power as the full-featured but very expensive true ICE). The ARM7TDMI core, for example, provides a core add-on known as EmbeddedICE, documented in the ARM ARM7TDMI Technical Reference Manual (Chapters 5 and Appendix B), as well as the Atmel ARM7TDMI datasheet. The ARMv7 architecture (including Cortex-M3) includes even more debug options like ETM, FPB, DWT. There are also vendor-specific embedded debug blocks, such as Freescale's Background Debug Mode (BDM).

ARMv7 Debugging Support

The ARMv7 architecture prescribes several on-chip debug modules used to aid in different aspects of testing and debugging.

Instrumentation Trace Macrocell (ITM)

The ITM can be used for logging events in real-time. It represents one or more FIFO's (implementation dependent) which can hold arbitrary data and send it off-chip through a TPIU (see below). For example, your scheduler could write to the ITM every time it does a context switch. The value written could be the number of the thread that runs next. In this way you can observe a timeline of thread execution that is much more lightweight than `printf()` calls.

The ITM also supports timestamped FIFO writes using a built-in counter so that the trace log contains both event and inter-event-time information. This feature can be used for profiling functions, i.e., seeing how long they take to execute (or how often they are called per unit time).

Data Watchpoint and Trace (DWT)

The DWT is a powerful hardware block that allows for much finer-grained profiling than logging through the ITM, as well as statistical sampling of the program counter (a coarser Monte Carlo approach to profiling). As an example of fine-grain profiling, the DWT provides 8-bit event counters for:

- total number of instruction cycles executed (since the counter was reset)
- additional cycles needed for load/store instructions
- how many instructions executed in 0 cycles (because they were folded in to other instructions)
- how many cycles were executed to support exception entry and return
- how many cycles were executed to support entry into a sleep mode

The other main purpose of the DWT is to support watchpoints: particular addresses in RAM or FLASH which should cause the CPU to halt and enter debug mode. Setting a breakpoint in GDB, for example, causes a particular address in FLASH (the address of the breakpoint) to be watched by the DWT. By watching addresses in RAM, you can tell the CPU to halt when a particular variable is read from or written to. This is a very handy way to watch for things like stack overflows or to see when/how variables "mysteriously" change their values.

Embedded Trace Macrocell (ETM)

An ETM is a much more complicated form of ITM that has supported for powerful multi-step tracing and counting operations. It essentially packages a logic analyzer onto the microcontroller. See the (476-page) ARM document IHI0014 for more details.

Trace Port Interface Unit (TPIU)

All of the data coming from the ITM, DWT, and ETM funnel their way through the TPIU onto their eventual destination of pins on the chip. Data from these trace/debug blocks is formatted in packets, and the packets are sent out of the TPIU in a serial format. The TPIU can be configured for the number of bits per word sent out, serial clock speed, and serial encoding (Manchester, NRZ, or synchronous with clock).

Flash Patch and Breakpoint Unit (FPB)

While the DWT can set a limited number of hardware breakpoints (by using comparators between the instruction address bus and the breakpoint location), what if you want to set a LOT of breakpoints? In the days when code executed from RAM, you could just replace a 16-bit/32-bit instruction with a special "breakpoint instruction" (BKPT in ARMv7-M) which, when executed, would return control to the debugger. Obviously, this can't be done with FLASH as you can't just overwrite a single memory location. The FPB solves this problem by remapping an entire block of FLASH memory to RAM so the code continues to look like it executes out of FLASH (the addresses are the same) but is in fact running out of RAM, and the replace-the-instruction-with-BKPT approach now works.

The ARM documents suggest you can use the FPB for in-the-field patches for code but this doesn't sound like it's a big improvement over just upgrading the code in FLASH.

System Control Block

Through register accesses in the SCB space you can halt the CPU and also single-step the CPU. You can also read and write any of the on-chip registers while the CPU is halted, as well as the MSP, PSP, xPSR, etc. So how exactly can you do these things if the CPU is halted? Accessing the debug-related registers in the SCB space is done through a special "back door" channel known as the Debug Access Port (or DAP). This can be physically accessed through a JTAG or single-wire debug (SWD) interface.

JTAG : The Original Series

The acronym JTAG stands for Joint Test Action Group and is a nickname for the committee that drafted "IEEE Standard 1149.1 - 1990 Standard Test Access Port and Boundary-Scan Architecture". The intention was to simplify testing the interconnection of chips on a PCB in an era of increasing pin density where traditional bed-of-nails testing became impractical.

In boundary scan testing, the chip can be halted and all of the hardware pins of the device (i.e., the chip's boundary to the outside world) can either be read out to a debugger, or forced to a given state (high, low, tri-state). Boundary scan can be used to test interconnection traces between chips (e.g., force pin 37 of IC1 to be a logic 1 and make sure that pin 69 of IC2, which is connected to pin 37 of IC1, also reads as a logic 1, thus verifying the connection between the two pins). Really, you're testing to see if the PCB was manufactured correctly, if there are any short circuits or open circuits, and if all chip pins are properly soldered to their pads.

A device that implements this standard must have:

- a pre-defined set of hardware pins with well-defined electrical properties
- an on-chip test access port (TAP) hardware block
- a pre-defined mechanism for accessing the on-chip TAP using these pins
- a pre-defined minimum set of TAP instructions for accessing the scan chains

Only 5 hardware pins are required, and one of them can be the chip's reset signal (`nRST`) so really only 4 additional pins are required in order to implement a "JTAG interface". These pins are:

- TCK - a clock signal provided by the external host PC. All communication with the TAP is done synchronously using this clock signal, which is entirely different from the "microcontroller clock".
- TDI - an input data signal (i.e., from host to microcontroller) which is latched on rising edges of TCK
- TDO - an output data signal (i.e., from microcontroller to host) which is driven in response to falling edges of TCK
- TMS - a "test mode select" input signal (i.e., from host to microcontroller) that is used in conjunction with TDI to give instructions to the TAP.

The TAP hardware is implemented as a simple 16-state state machine (Figure 1), thus adding minimal overhead to the microcontroller silicon. The TMS signal is used to cause state-to-state transitions. Data is scanned into or out of the TAP by remaining in the same state and providing multiple TCK clocks while driving TDI (to scan data in) or reading TDO (to scan data out). The instructions implemented by the ARM TAP are summarized in Figure 2.

Boundary scan can also be used to perform limited debugging, but only when the code to be debugged is somehow reflected in the state of the device's physical pins. In "the old days", all of a microcontroller's ROM and RAM were external hence all instructions and memory accesses were externally snoopable on physical pins using a logic analyzer. But since the ARM core, FLASH, and RAM are all internal to the chip, boundary scan cannot be used to set breakpoints, for example, since the address bus is not reflected in the device's external pins.

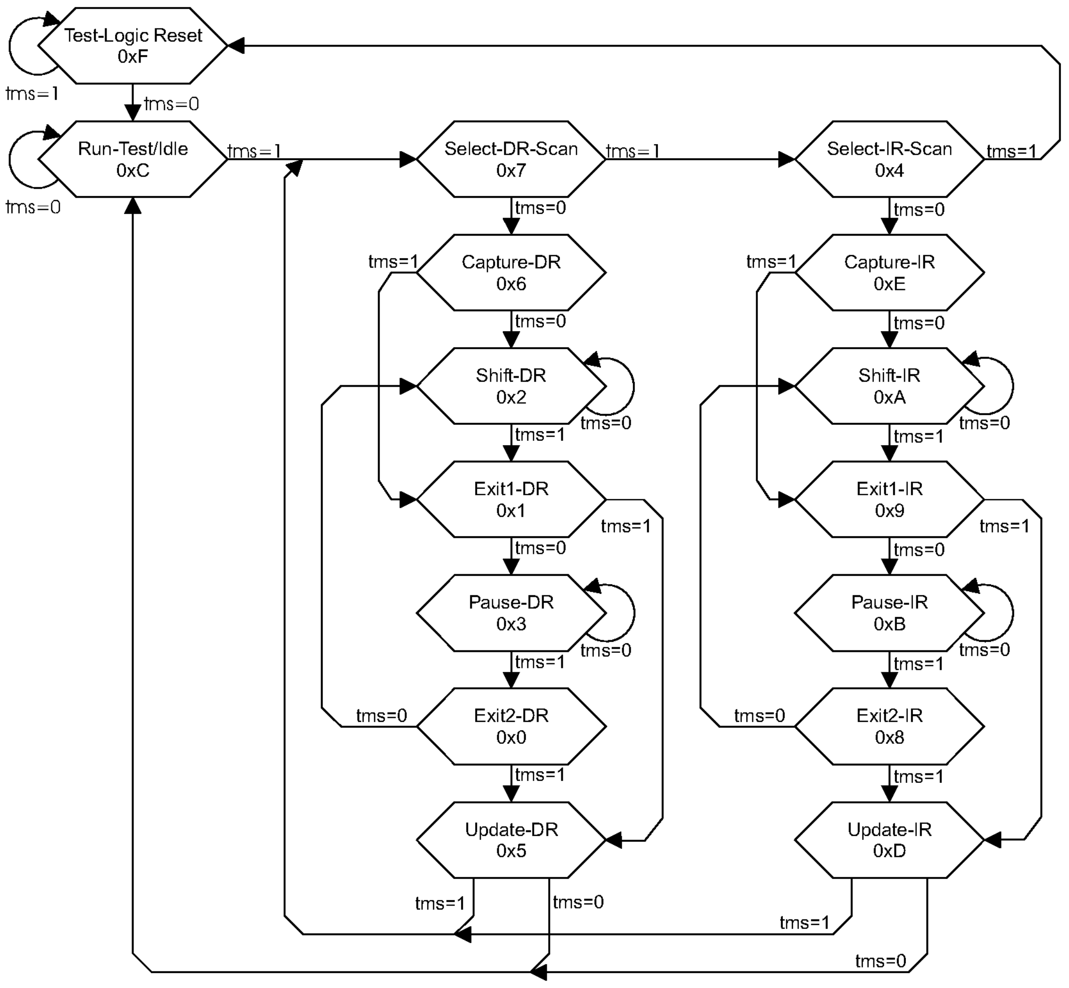

Figure 1: The TAP controller state machine effects state transitions on rising TCK edges and uses the TMS pin level to select the sequence of states.

The TAP can also access the internal scan chains of the microcontroller. A *scan chain* is a daisy-chained interconnection of on-chip signal storage elements, such as the address bus latch, data bus latch, control signals latch, etc. If you put the address bus latch (32 bits), data bus latch (32 bits), and control signals latch (41 bits) together side-by-side you would have a 105-bit scan chain (this is known as scan chain #0 on the AT91SAM7S). By stopping the processor and scanning out this scan chain, one bit at a time, you can deduce exactly what instruction the chip is executing and at what address, along with other information gleaned from the control signals such as whether this is a word, half-word, or byte-sized access, an instruction or data access, etc. Figure 4 illustrates how an internal output signal can be converted to a scan-enabled signal by the addition of a little bit of extra logic. The shift-in and shift-out signals of scan cells are connected in series to form a complete scan chain.

Scan chain #1 on the AT91SAM7S is a 33-bit subset of the full 105-bit scan chain #0, and contains only the 32-bit data bus plus a debug bit.

JTAG : The Next Generation

Straying away from its original intention, JTAG has become a common way to provide access to the ARM core's debugging resources (DAP, TPIU, ETM, ITM, DWT, etc.) and also to program the FLASH on the device. Simply by adding JTAG instructions and presenting the internal debug registers as components of a scan chain, the 5-pin JTAG interface can be used as the necessary back door for debug functions even when the CPU is halted.

Instruction	Binary	Hexadecimal
EXTEST	b0000	0x0
SCAN_N	b0010	0x2
SAMPLE/PRELOAD	b0011	0x3
RESTART	b0100	0x4
CLAMP	b0101	0x5
HIGHZ	b0111	0x7
CLAMPZ	b1001	0x9
INTEST	b1100	0xC
IDCODE	b1110	0xE
BYPASS	b1111	0xF

Figure 2: TAP instructions supported by the ARM TAP. For descriptions of these instructions, see the ARM7TDMI Technical Reference Manual.

JTAG Test Environment

The components necessary for in-circuit debugging using JTAG are:

- A physical device to drive the TCK/TDI/TDO/TMS pins of the microcontroller and connect back to the host computer (often called the *protocol converter*)
- A control program running on the host computer that takes care of all the complications of the TAP state machine and scan chains.

This is where things get nasty. Companies like to make money by selling you proprietary solutions to complicated situations. IAR, for example, sells the Embedded Workbench C/C++ compiler suite ($2995) for the ARM that includes an integrated debugger with support for JTAG debugging using a Segger "J-Link" USB-to-JTAG interface device ($299).

We can't even write our own control software (as complicated as it may be) for the J-Link since it is a proprietary device with an unpublished protocol.

To the rescue come the following:

- A variety of parallel-port-to-JTAG or USB-to-JTAG interface devices with an "open" interface format. These come in a variety of prices, the lowest of which is currently $20.95 for the Olimex ARM-JTAG parallel-port device. A $51.95 USB-based device (Olimex ARM-USB-TINY) is also available. Competing devices such as Macgraigor's Wiggler or Amontec's JTAGKey are more than $100 each.
- An amazing piece of free software known as OpenOCD, a control program for On-Chip Debugging using JTAG interface devices. This software hides all of the complexities of TAP controllers, scan chains, etc. and can work with a variety of protocol converters, including ARM-JTAG, ARM-USB-TINY, Wiggler, JTAGKey, and others.

Figure 5 illustrates all of the components that participate in an in-circuit debugging session.

The primary interface seen by the user is GDB (and/or its graphical cousin, Insight). We tell GDB to connect to a "remote debug server". This remote server can, of course, be located on our computer, but in general can be located on any network-connected computer, since a standard TCP/IP connection is used to connect to the server. This is what we are achieving when we type:

```
target remote localhost:3333
```

The above line says to connect to the computer named "`localhost`" (i.e., the computer you're already on) on port 3333 but it could just as easily have been a computer halfway around the world:

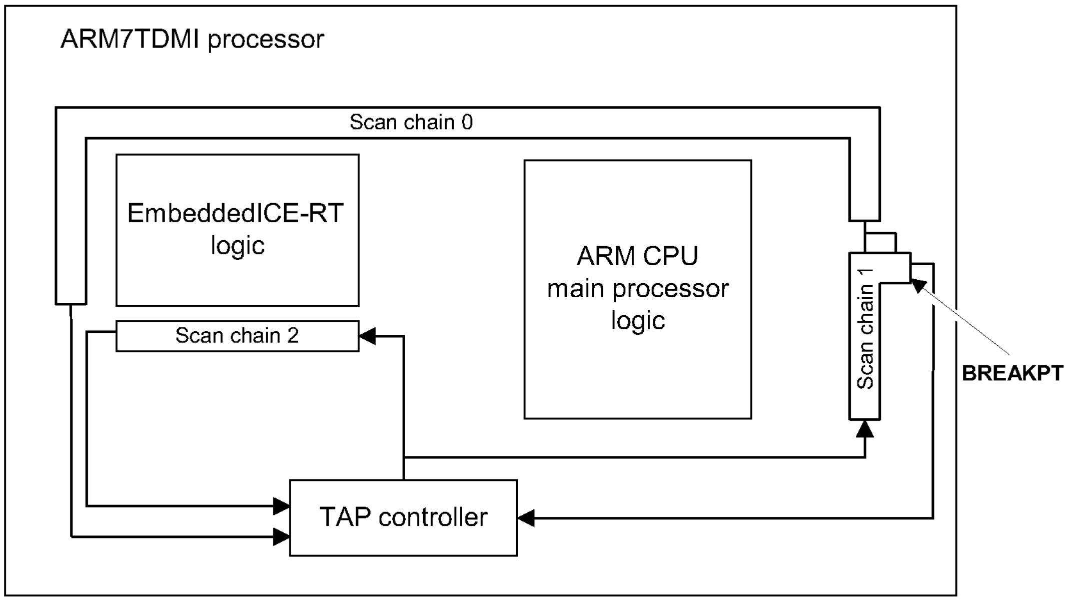

Figure 3: The TAP controller provides access to the on-chip scan chains, including the EmbeddedICE module, as well as to the boundary scan chain (not shown).

```
target remote 172.16.25.3:3333
```

The remote debug server is responsible for accepting GDB commands (e.g., set breakpoint, single step, display registers, etc.) and communicating with the target microcontroller's TAP controller. This server, OpenOCD in our case, must understand the TAP state machine, the target TAP instructions, the format of the target's scan chains, the behavior of the target's embedded debug modules, etc. thus is a very significant piece of software[1].

OpenOCD connects over either a parallel port, serial port, USB port, etc. to the JTAG protocol converter. This converter accepts simple commands and manipulates the TCK, TDI, etc. pins in response to these commands.

Note carefully that the JTAG protocol converter must either be electrically isolated from the target hardware or must share a common ground reference with the host PC. Both USB and serial port connections from the host PC to the ARM's USB/serial peripherals will establish a common ground between the ARM and the host PC, thus a non-isolated protocol converter is OK. But be careful in the future...don't ignore the nature of the power supplies in the system and pay attention to where the ground return signal paths are.

Notes

The JTAG interface shows up on many microcontrollers, CPLD's, and FPGA's today and shows promise of uniting the variety of on-chip debug (and in-system programming) solutions currently available[2]. This is good for developers as it minimizes the number of different software tools that developers have to master (as well as the number of dongles they have to not lose). Unfortunately, much of the software support for the chip operations continues to remain secret, so "a JTAG port" is by no means enough to use whatever hardware/software you have to work with a new microcontroller.

For example, although Atmel's AT91SAM7S devices have an "open" instruction set for JTAG manipulations (because it's prescribed by ARM), their AVR devices do not, even though they do have a JTAG interface (look carefully at your MW2/Ninja boards...they have 4 pins on Port C called TCK/TMS/TDI/TDO). The TAP state machine on an ATmega324P is the same as on all other microcontrollers, but the equivalent to the ITM/DWT module is proprietary. How do you halt the AVR core through the TAP? Only Atmel knows. How do you inspect the hardware registers? Only Atmel knows. So how do you do in-circuit hardware debugging of the ATmega324P? You buy Atmel's AVR-JTAG-ICE for $299.

[1]Send your thanks to Dominic Raith, who wrote the first draft of this software as part of his Master's thesis.

[2]...unless it doesn't. ARM is now favoring the single-wire debug (SWD) interface and vendors like NXP are already removing JTAG ports from their ARM processors and leaving only SWD ports. Standardization is good for consumers, but bad for vendors who can make much more money on proprietary and non-commoditized solutions.

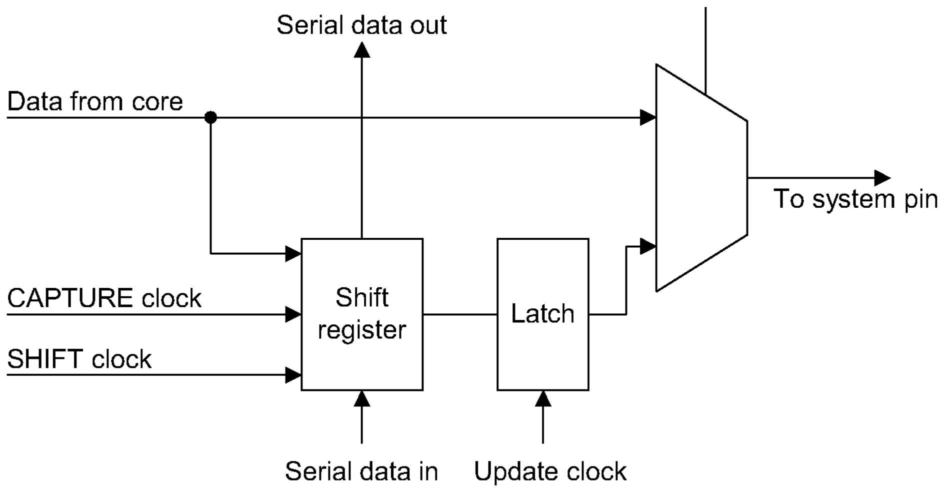

Figure 4: A scan cell is formed by adding a shift register, latch, and multiplexer to an existing signal. The extra circuitry can either pass the internal signal through to the output, capture the signal in the scan latch (for later scan-out), or apply the stored latch value (from a previous scan-in) to the output.

Integrated Protocol Converters

The protocol converter is built in to the TI LM3S6965 development kit, illustrated in Figure 6. The design obviates the need for a separate JTAG protocol converter ($299 back in your pocket) since a single FTDI FT2232C USB-to-serial interface chip provides both data connectivity and can be used to manipulate the TAP pins on the microcontroller (the FT2232C has lots of extra I/O pins).

The necessary pulsing of TCK/TMS/TDI etc. is incorporated inside the integrated development environment that comes with the evaluation board, so there is really nothing needed to do in-circuit hardware debugging besides the board itself and a USB cable. It doesn't get any simpler.

Most of the time, this setup Just Works, and is a nice change from separate power supplies, dongles, extra cables, etc. However...the Keil IDE (as an example) is a "code-limited version" (32k) and has a lot of magic built in, so when things just DON'T work, you're on your own[3].

[3]Like, for example, when you discover that "32k code-limited version" really means text+data+bss must be less than 32k, so it doesn't matter that your code is tiny, you can't map the microcontroller's RAM into an array as that increases bss beyond the 32k limit. But I guess "32k text+data+bss limited version" doesn't look so good on marketing literature.

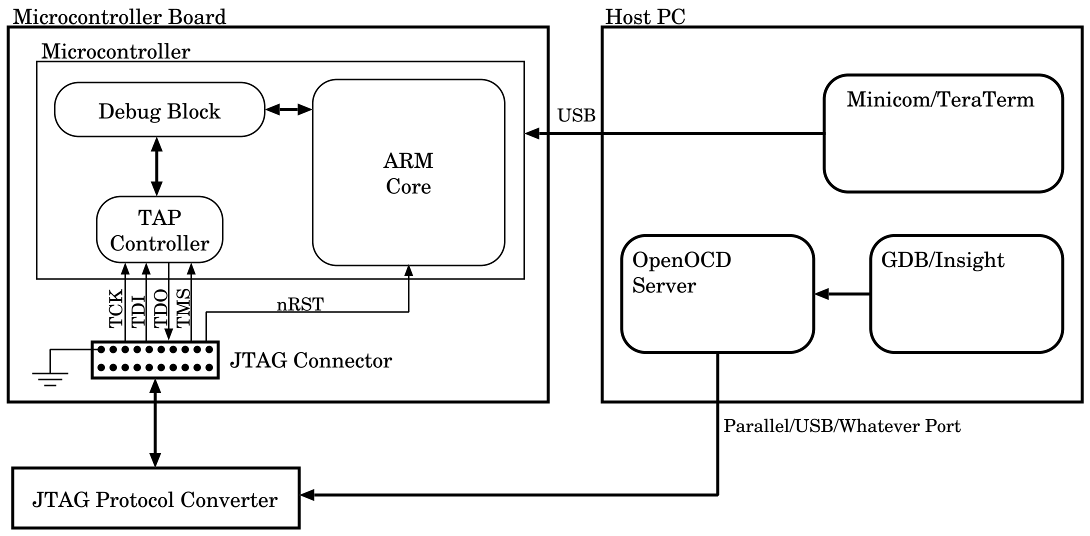

Figure 5: A JTAG-based in-circuit debugging session requires a debugger (GDB/Insight), a JTAG protocol server (OpenOCD), and a JTAG protocol converter (Wiggler, JTAGKey, ARM-JTAG, etc.). A normal connection between the user and the ARM controller (using TeraTerm, minicom, etc.) can still be used. The JTAG protocol converter must either be electrically isolated or share a common ground with the target hardware.

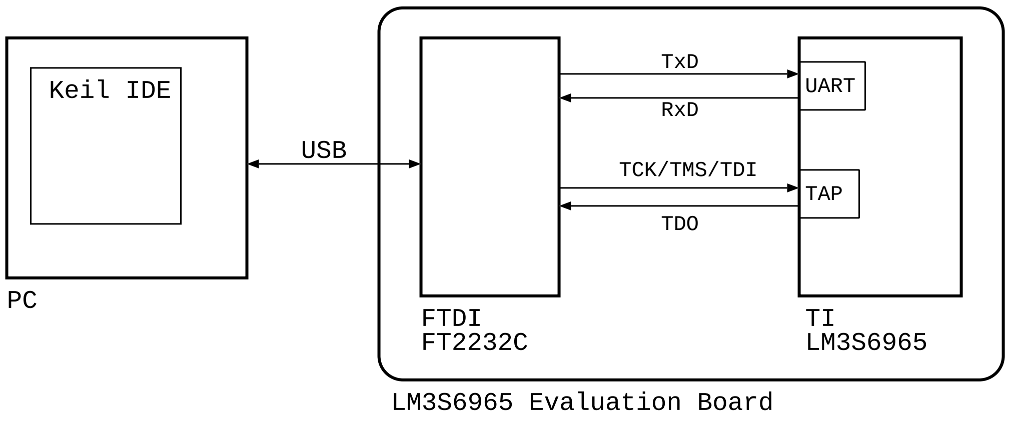

Figure 6: The TI LM3S6965 development board incorporates the JTAG protocol converter in the FT2232C USB-to-serial converter, thus simplifying the process of debugging. The Keil integrated development environment (or CodeSourcery, or other toolchain) simplifies things further by including the necessary JTAG machinery. A USB cable and the development board itself are the only components of the development environment.