

THE AMAZING \$1 MICROCONTROLLER

A new series that explores 21 different microcontrollers — all less than \$1 — to help familiarize you with all the major ecosystems out there.

Texto completo em: <https://jaycarlson.net/microcontrollers/>

As an embedded design consultant, the diverse collection of projects on my desk need an equally-diverse collection of microcontroller architectures that have the performance, peripheral selection, and power numbers to be the backbone of successful projects. At the same time, we all have our go-to chips – those parts that linger in our toolkit after being picked up in school, through forum posts, or from previous projects.

In 2017, we saw several new MCUs hit the market, as well as general trends continuing in the industry: the migration to open-source, cross-platform development environments and toolchains; new code-generator tools that integrate seamlessly (or not so seamlessly...) into IDEs; and, most notably, the continued invasion of ARM Cortex-M0+ parts into the 8-bit space.

I wanted to take a quick pulse of the industry to see where everything is – and what I've been missing while backed into my corner of DigiKey's web site.

It's time for a good ol' microcontroller shoot-out.

THE RULES

While some projects that come across my desk are complex enough to require a hundreds-of-MHz microcontroller with all the bells and whistles, it's amazing how many projects work great using nothing more than a \$1 chip – so this is the only rule I established for this microcontroller review. To get technical: I purchased several different MCUs – all less than a \$1 – from a wide variety of brands and distributors. I'm sure people will chime in and either claim that a part is more than a dollar, or that I should have used another part which can be had for less than a dollar. I used a price-break of 100 units when determining pricing, and I looked at typical, general suppliers I personally use when shopping for parts – I avoided eBay/AliExpress/Taobao unless they were the only source for the parts, which is common for devices most popular in China and Taiwan.

I wanted to explore the \$1 pricing zone specifically because it's the least amount of money you can spend on an MCU that's still general-purpose enough to be widely useful in a diverse array of projects.

Any cheaper, and you end up with 6- or 8-pin parts with only a few dozen bytes of RAM, no ADC, nor any peripherals other than a single timer and some GPIO.

Any more expensive, and the field completely opens up to an overwhelming number of parts – all with heavily-specialized peripherals and connectivity options.

These MCUs were selected to represent their entire families – or sub-families, depending on the architecture – and in my analysis, I'll offer some information about the family as a whole.

If you want to scroll down and find out who the winner is, don't bother – there's really no sense in trying to declare the “king of \$1 MCUs” as everyone knows the best microcontroller is the one that best matches your application needs. I mean, everyone knows the best microcontroller is the one you already know how to use. No, wait – the best microcontroller is definitely the one that is easiest to prototype with. Or maybe that has the lowest impact on BOM pricing?

I can't even decide on the criteria for the best microcontroller – let alone crown a winner.

What I will do, however, is offer a ton of different recommendations for different users at the end. Read on!

THE CRITERIA

This microcontroller selection guide will have both qualitative and quantitative assessments. Overall, I'll be looking at a few different categories:

PARAMETRICS, PACKAGING, AND PERIPHERALS

Within a particular family, what is the range of core speed? Memory? Peripherals? Price? Package options?

Some microcontroller families are huge – with hundreds of different models that you can select from to find the perfect MCU for your application. Some families are much smaller, which means you're essentially going to pay for peripherals, memory, or features you don't need. But these have an economies-of-scale effect; if we only have to produce five different MCU models, we'll be able to make a lot more of each of them, driving down the price. How do different MCU families end up on that spectrum?

Package availability is another huge factor to consider. A professional electronics engineer working on a wearable consumer product might be looking for a wafer-level CSP package that's less than 2×2 mm in size. A hobbyist who is uncomfortable with surface-mount soldering may be looking for a legacy DIP package that can be used with breadboards and solder protoboards. Different manufacturers choose packaging options carefully, so before you dive into an architecture for a project, one of the first things to consider is making sure that it's in a package you actually want to deal with.

Peripherals can vary widely from architecture to architecture. Some MCUs have extremely powerful peripherals with multiple interrupt channels, DMA, internal clock generators, tons of power configuration control, and various clocking options. Others are incredibly simple – almost *basic*. Just as before, different people will be looking for different things (even for different applications). It would be a massive undertaking to go over every single peripheral on these MCUs, but I'll focus on the ones that all MCUs have in common, and point out fine-print "gotchas" that datasheets always seem to glance over.

DEVELOPMENT EXPERIENCE

Any microcontroller review or selection guide should include a discussion of the overall development environment and experience.

While this is where things get subjective and opinion-oriented, I'll attempt to present "just the facts" and let you decide what you care about. The main source of subjectivity comes from *weighing* these facts appropriately, which I will not attempt to do.

IDEs / SDKs / Compilers: What is the manufacturer-suggested IDE for developing code on the MCU? Are there other options? What compilers does the microcontroller support? Is the software cross-platform? How much does it cost? These are the sorts of things I'll be exploring while evaluating the software for the MCU architecture.

Platform functionality and features will vary a lot by architecture, but I'll look at basic project management, source-code editor quality, initialization code-generation tools, run-time peripheral libraries, debugging experience, and documentation accessibility.

I'll focus on manufacturer-provided or manufacturer-suggested IDEs and compilers (and these will be what I use to benchmark the MCU). There are more than a dozen compilers / IDEs available for many of these architectures, so I can't reasonably review all of them. Feel free to express your contempt of my methodology in the comments section.

Programmers / debuggers / emulators / dev boards: What dev boards and debuggers are available for the ecosystem? How clunky is the debugging experience? Every company has a slightly different philosophy for development boards and debuggers, so this will be interesting to compare.

PERFORMANCE

I've established three different code samples I'll be using to benchmark the parts in this microcontroller review; I'll be measuring quantitative parameters like benchmark speed, clock-cycle efficiency, power efficiency, and code-size efficiency.

WHILE(1) BLINK

For this test, I'll toggle a pin in a `while()` loop. I'll use the fastest C code possible – while also reporting if the code generator tool or peripheral libraries were able to produce efficient code. I'll use bitwise complement or GPIO-specific “toggle” registers if the platform supports it, otherwise, I'll resort to a read-modify-write operation. I'll report on which instructions were executed, and the number of cycles they took.

What this tests: This gives some good intuition for how the platform works – because it is so low-level, we'll be able to easily look at the assembly code and individual instruction timing. Since many of these parts operate above flash read speed, this will allow us to see what's going on with the flash read accelerator (if one exists), and as a general diagnostic / debugging tool for getting the platform up and running at the proper speed. This routine will obviously also test bit manipulation performance (though this is rarely important in general-purpose projects).

64-SAMPLE BIQUAD FILTER

This is an example of a real-world application where you often need good, real-time performance, so I thought it would be a perfect test to evaluate the raw processing power of each microcontroller. For this test, I'll process 16-bit signed integer data through a 400 Hz second-order high-pass filter (Transposed Direct Form II implementation, for those playing along at home), assuming a sample rate of 8 kHz. We won't *actually* sample the data from an ADC – instead, we'll process 64-element arrays of dummy input data, and record the time it takes to complete this process (by wiggling a GPIO pin run into my [500 MHz Saleae Logic Pro 16 logic analyzer](#)).

In addition to the samples-per-second measure of raw processing power, I'll also measure power consumption, which will give us a “nanjoule-per-sample” measure; this will help you figure out how *efficient* a processor is. While I've traditionally used [µCurrent](#) units for this,

I ended up using a [Silicon Labs EFM8 Sleepy Bee STK](#) – I ripped the target processor off the board, turning it into a \$30 time-domain logarithmic power meter. If you're interested in more information, check out my [EFM8](#) review, which has the details of these excellent tools.

What this tests: Memory and 16-bit math performance per microamp, essentially. The 8-bit MCUs in our round up are going to struggle with this pretty hardcore – it'll be interesting to see just how much better the 16 and 32-bit MCUs do. Like it or hate it, this will also evaluate a compiler's optimization abilities since different compilers implement math routines quite differently.

DMX-512 RGB LIGHT

DMX-512 is a commonly-used lighting protocol for stage, club, and commercial lighting systems. Electrically, it uses RS-485; the protocol uses a long BREAK message at the beginning, followed by a "0" and then 511 bytes of data, transmitted at 250 kbaud. In this test, I'll implement a DMX-512 receiver that directly drives a common-anode RGB LED. I will do this with whatever peripheral library or code-generator tool the manufacturer provides (if any at all).

While you should really look for a precisely-timed break message, since this is only for prototyping, I'll detect the start-of-frame by looking for an RX framing error (or a "break" signal, as some UARTs support LIN).

I'll minimize power consumption by lowering the frequency of the CPU as much as possible, using interrupt-based UART receiver routines, and halting or sleeping the CPU. I'll report the average power consumption (with the LED removed from the circuit, of course). To get a rough idea of the completeness and quality of the code-generator tools or peripheral libraries, I'll report the total number of statements I had to write, as well as the flash usage.

What this tests: This is a sort of holistic test that lets me get into the ecosystem and play around with the platform. This stuff is the bread and butter of embedded programming: interrupt-based UART reception with a little bit of flair (framing error detection), multi-channel PWM configuration, and nearly-always-halted state-machine-style CPU programming. Once you have your hardware set up and you know what you're doing (say, after you've implemented this on a dozen MCUs before...), with a good code-gen tool or peripheral library, you should be able to program this with just a few lines of code in an hour or less – hopefully without having to hit the datasheet much at all.

Test notes: I'm using [FreeStyler](#) to generate DMX messages through an FTDI USB-to-serial converter (the program uses the [Enttec Open DMX](#) plugin to do this). As my FTDI cable is 5V, I put a 1k resistor with a 3.3V zener diode to to ground, which clamps the signal to 3.3V. The

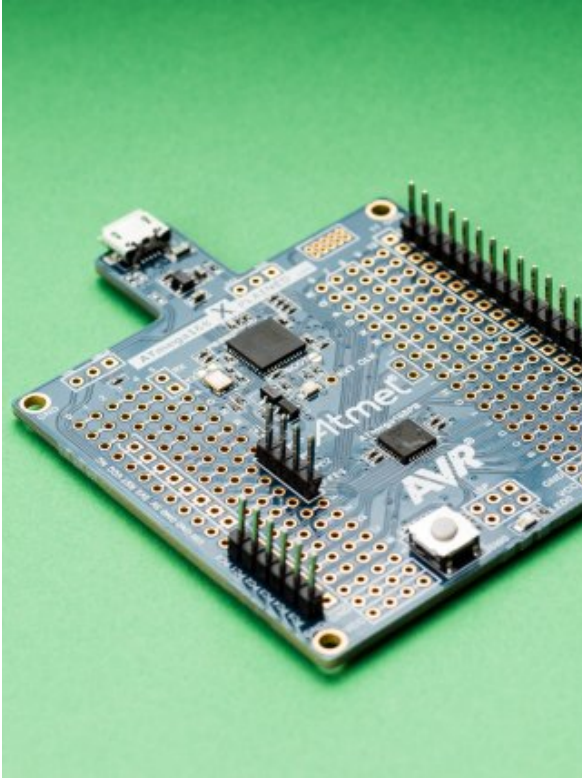
zener clamp isn't there to protect the MCU – all of these chips tested have diode catches to protect from over-voltage – but rather, so that the MCU doesn't inadvertently draw power from the Rx pin, which would ruin my current measurement results.

EVALUATION CODE

All code I used to evaluate these parts in this microcontroller selection guide is available on my [microcontroller-test-code GitHub repo](#). Check it out, fork it, submit patches, and keep me honest – if your results differ from mine, let's get this post updated.

THE CONTENDERS

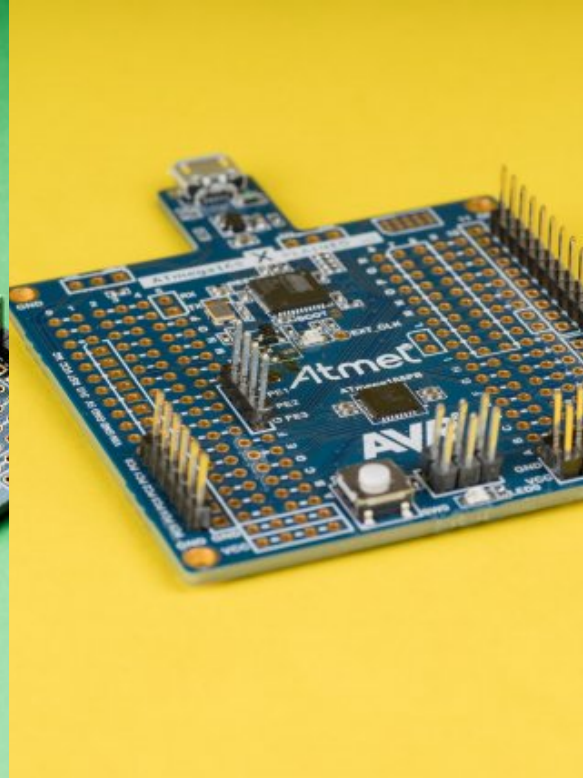
This page will compare the devices, development tools, and IDEs all together. However, to prevent this article from getting overwhelmingly long, I've created review pages for each device that cover way more details about the architecture – along with more complete testing notes, different benchmarking variations, and in-depth assessment. If an architecture strikes your interest, you should definitely check out the full review below.



ATMEL TINYAVR

Part: ATtiny1616

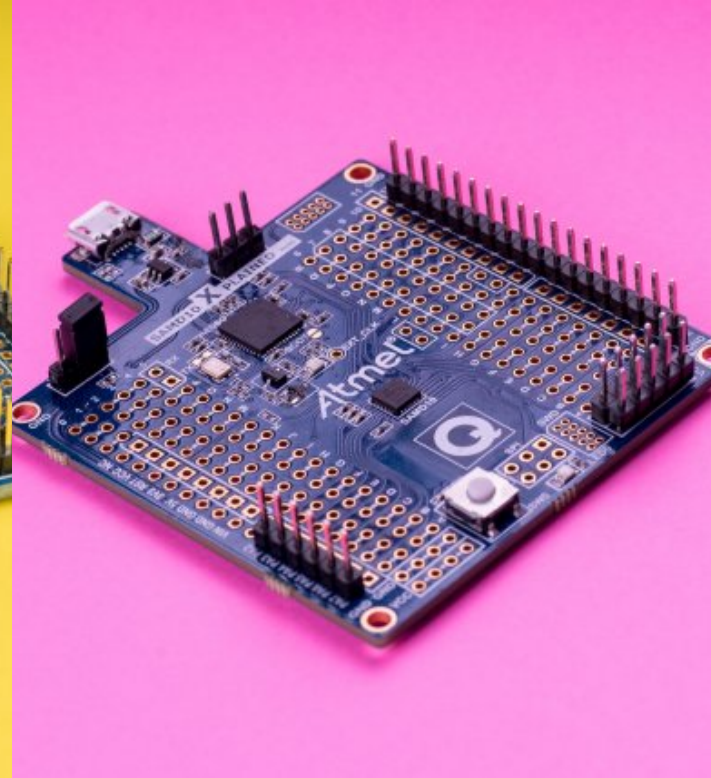
The new-for-2017 tinyAVR line includes seven parts with XMEGA-style peripherals, a two-cycle 8-bit multiplier, the new UPDI one-wire debug interface, and a 20 MHz oscillator that should shoot some energy into this line of entry-level AVR controllers that was looking quite long in the tooth next to other 8-bit parts.



ATMEL MEGA AVR

Part: ATmega168PB

The AVR earned its hobbyist-friend badge as the first MCU programmed in C with open-source tools. The new version of the classic ATmega168 takes a price cut due to a die-shrink but little else has changed, including the anemic 8 MHz internal oscillator — and, like the tinyAVR, must sip 5V to hit its full 20 MHz speed.



ATMEL SAM D10

Part: ATSAMD10D14A

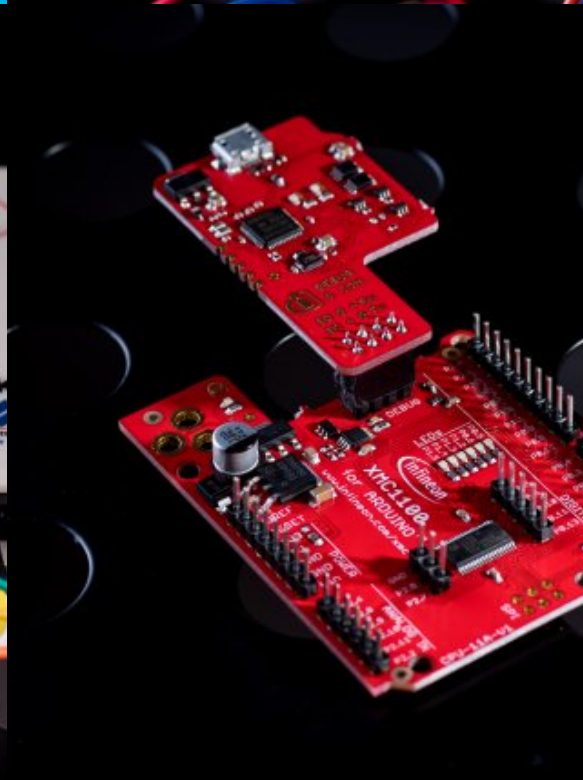
Atmel is positioning their least-expensive ARM Cortex-M0 offering — the new SAM D10 — to kill off all but the smallest TinyAVR MCUs with its performance numbers, peripherals, and price. Stand-out analog peripherals capstone a peripheral set and memory configuration that endows this part with good value.



HOLTEK HT-66

Part: HT66F0185

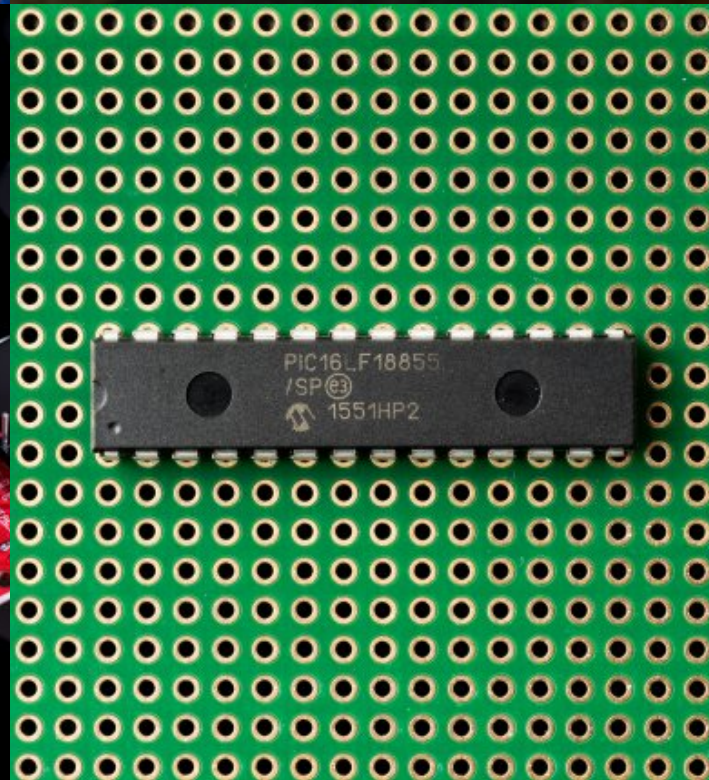
A basic 8-bit microcontroller with a slow, 4-cycle PIC16-style single accumulator core. An anemic peripheral selection and limited memory capacity makes this a better one-trick pony than a main system controller. Holtek has a wide range of application-specific MCUs that integrate this core with HV power and other goodies.



INFINEON XMC1100

Part: XMC1100T016X0016

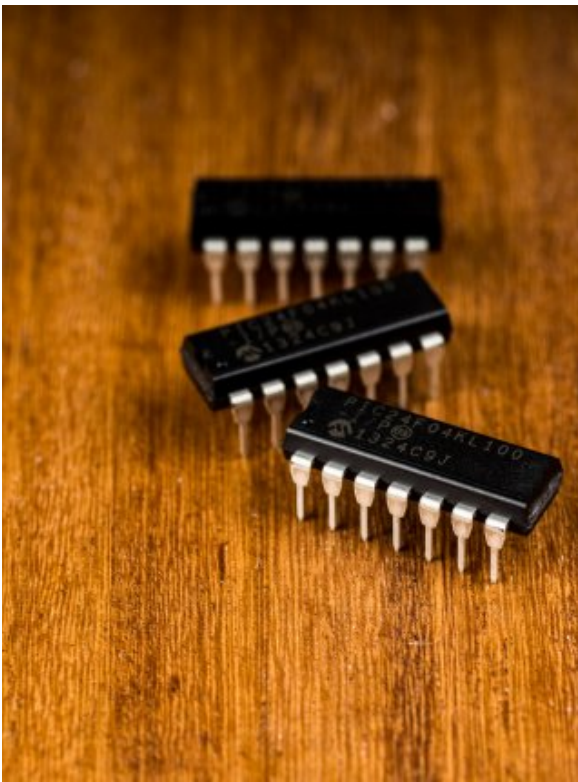
Infineon Arm chips are common picks for control projects, and the new XMC1100 is no different. With 16K of RAM, a 1 MSPS six-channel ADC, flexible communications, up to 16 timer capture channels, and the ability to form a 64-bit timer for large-range timing gives this paragon a bit of personality among entry-level Cortex-M0 microcontrollers.



MICROCHIP PIC16

Part: PIC16LF18325

Competing with the 8051 as the most famous microcontroller of all time, the latest PIC16 Five-Digit Enhanced parts feature improved peripheral interconnectivity, more timers, and a better analog. Still driven by a sluggish core that clammers along at one-fourth its clock speed, the PIC16 has always been best-suited for peripheral-heavy workloads.



MICROCHIP PIC24

Part: PIC24F04KL100

An expensive 16-bit part that's designed (and priced) to mirror the MSP430. While it has decent performance and power consumption, it's hard not to look toward other parts — especially the PIC32MM — which offer better pricing, and can beat the PIC24 on everything other than deep-sleep current consumption.

MICROCHIP PIC32MM

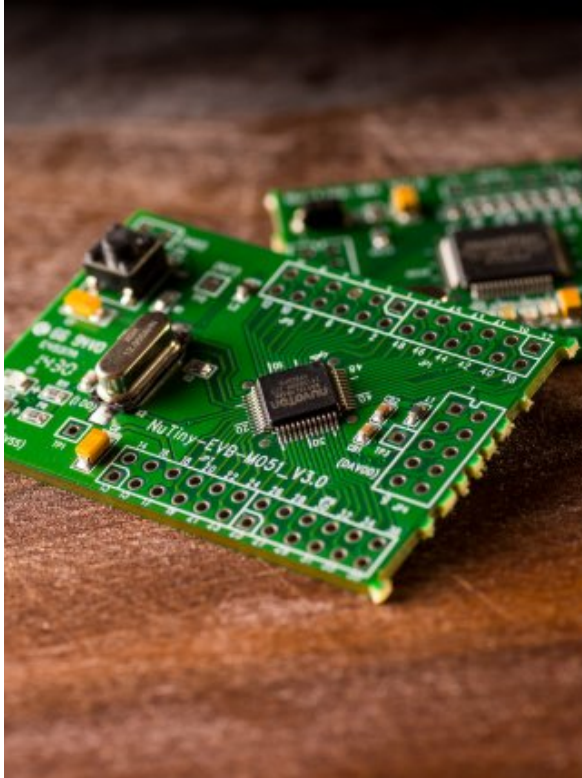
Part: PIC32MM0064

The 32-bit MIPS-powered PIC32MM compares similarly with ARM controllers on a per-cycle basis, but doesn't provide the same flexibility with tooling that ARM does. It's a great part for 32-bit beginners though, as it brings along PIC18/PIC24-style peripherals and a fuse-based configuration, easing development.

NUVOTON N76

Part: N76E003AT20

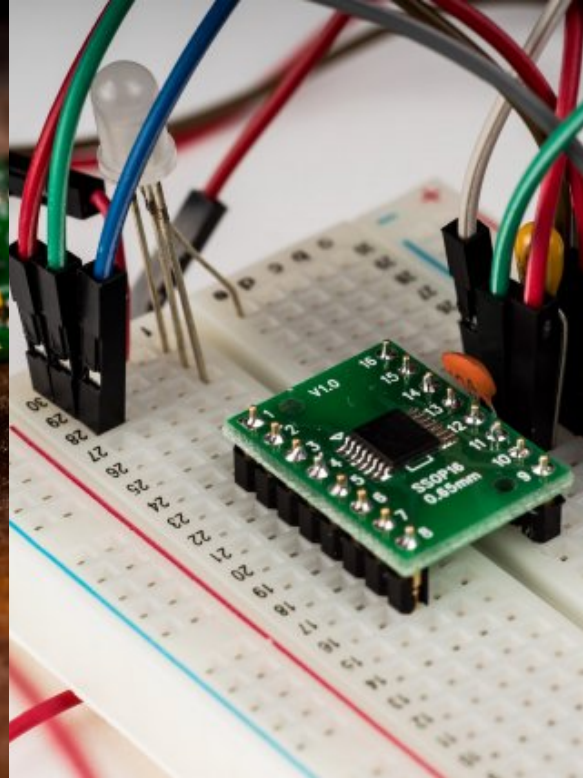
The N76 is a 1T-style 8051 that brings a few twists and useful additions to the basic set of '51 peripherals. This MCU has a slower instruction timing versus the EFM8 or STC8, but it's hard to complain about a well-documented, fully-featured MCU with North American support that you can buy with East Asia pricing.



NUVOTON M051

Part: M052LDN

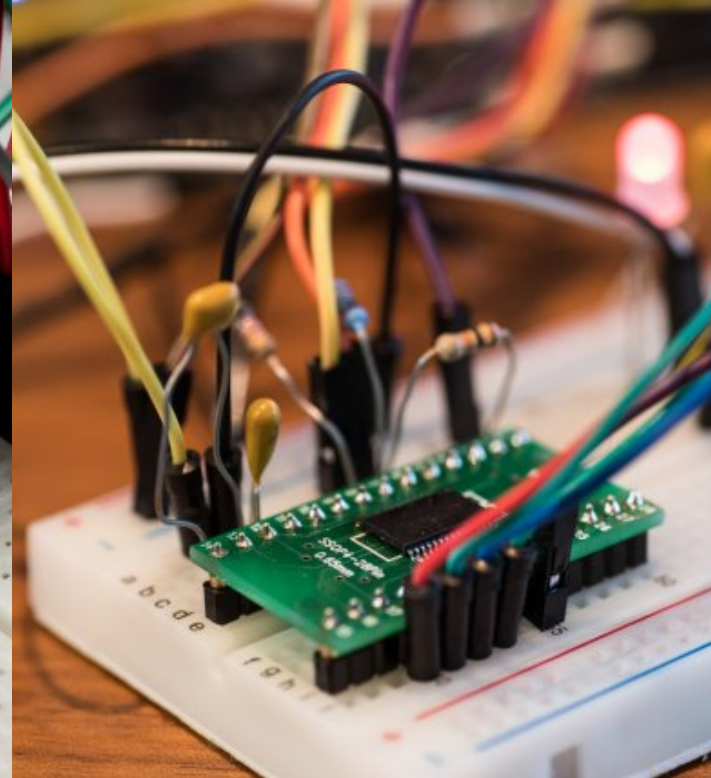
The M051 series is a high-value, 48 MHz Cortex-M0 with excellent timers and comms peripherals, coherent, easy-to-use functions, an oriented peripheral library, a relatively high pin-count, and utilitarian dev tools. The Achilles heel is the somewhat-limited I/O options, buggy software, and high power consumption figures.



NXP LPC811

Part: LPC811M001JDH16

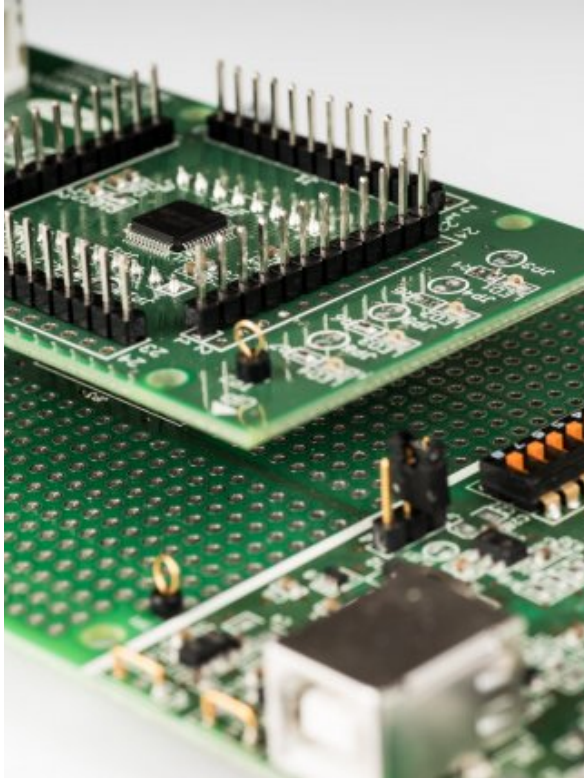
The LPC81x is famous among hobbyists for the LPC810 — an 8-pin DIP-package MCU. For everyone else, the LPC81x is an older, forgettable 30 MHz ARM that's slow on peripherals (it doesn't even have an ADC). An easy-to-use function-oriented peripheral library, serial loader, and plenty of code examples on blog posts keep this part alive.



RENESAS RL-78

Part: R5F102A8ASP

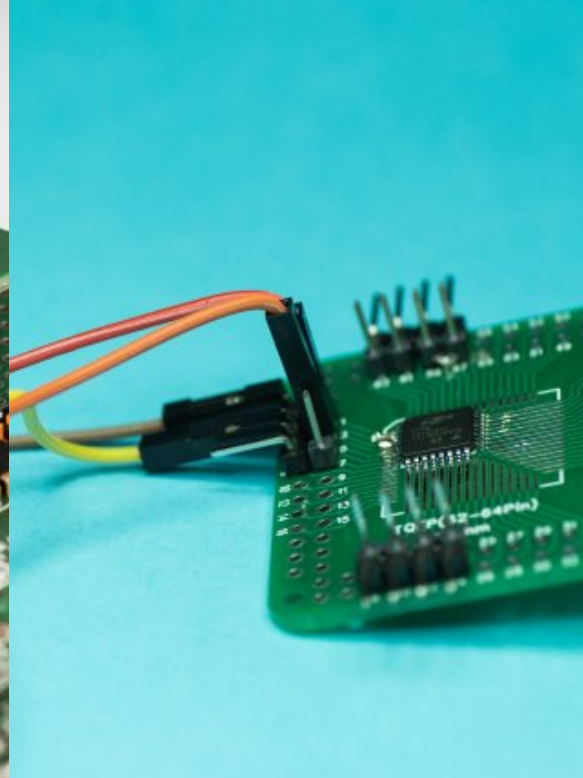
With the RL-78, Renesas built a clever hybrid MCU with an 8-bit-wide data path and a 16-bit-wide ALU, balancing cost and performance. Excellent low-power consumption, arrayed comms and timer peripherals, plus a good code-gen tool built into the free Eclipse IDE makes this part a strong competitor against the PIC24 and MSP430.



SANYO LC87

Part: LC87F1M16

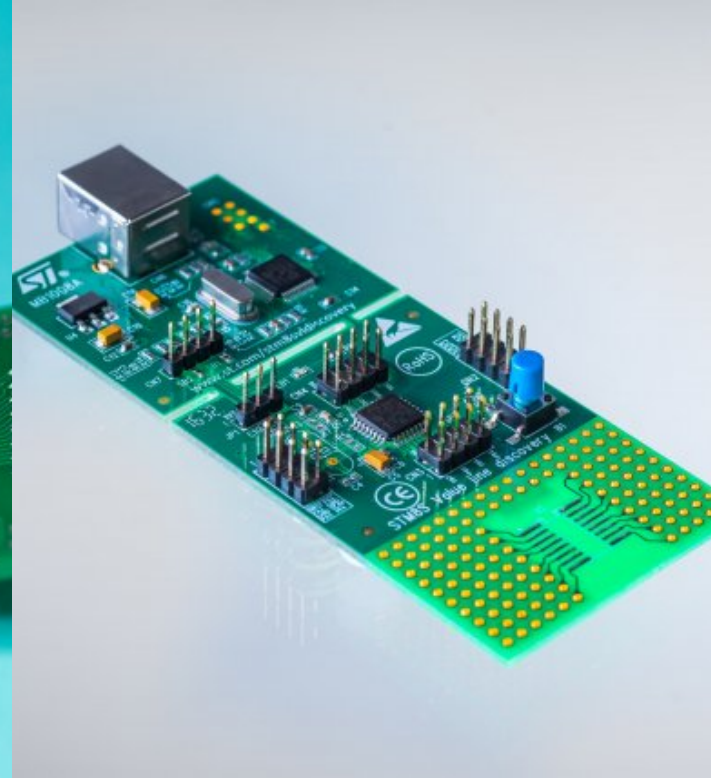
There's not much to like in the LC87. Abysmal power consumption, lackluster peripherals, unfriendly pricing, and an obnoxiously antiquated development ecosystem should steer almost any rational person from this architecture that, from the copyright dates on the development tool, looks to be headed to the grave.



SILICON LABS EFM8

Part: EFM8LB11

The EFM8 Laser Bee is a snappy 10-MHz 8051 MCU that's both the fastest 8-bit MCU in our round-up as well as one of the lowest-power. Low-cost tools, a free cross-platform Eclipse-based IDE, and a slew of easy-to-program peripherals should get both professionals and hobbyists interested in exploring this platform.



ST STM8

Part: STM8S103F3P6

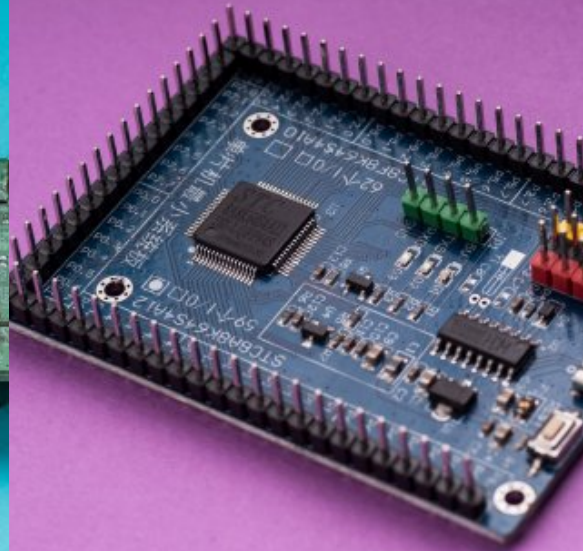
The STM8 feels like an ARM in disguise: a 32-bit-wide program memory bus with efficient computing performance, peripheral power gating, and a nested vector interrupt controller makes this thing look more like its STM32 big brothers. If only its STVD development environment felt as modern as its peripheral set does.



ST STM32F0

Part: STM32F030F4P6

While the F0 has an average peripheral set and worse-than-average power consumption, its low-cost ST-Link debugger, free IDE, good code-gen tools, and huge parametric latitude (up to the 18 MHz, 2 MB STM32F4) make this useful family to learn — plus everyone seems to have an STM Discovery board laying around



STCMICRO STC8

Part: STC8A8K64S4A12

A brand-new, single-cycle 8051 jam-packed full of flash, RAM, and oodles of peripherals — and a large, 64-pin package to make use of all these guts. Unfortunately, this part isn't quite ready for prime-time: the datasheet hasn't been translated into English yet, the errata is massive, and there's limited availability of the part.



TI MSP430

Part: MSP430FR2111

Texas Instruments dials down the power consumption in the latest iteration of the MSP430. FRAM memory, flexible power states, and tons of internal clocking options make this part a battery's dream come true. You'll pay for this power, though: the MSP430 can be twice as expensive as many 8-bit parts.

SPECS COMPARISON

CORE

Microcontrollers continue to divide into two camps – those with vendor-specific core architectures, and those who use a third-party core design. Out of the 21 microcontrollers reviewed here, eight of them use a 32-bit ARM core, which is becoming ubiquitous in the industry – even at this price point. Three of the microcontrollers use an 8-bit 8051-compatible ISA. The remaining ten use the vendor's proprietary core design: six are 8-bit parts, three are 16-bit parts, and the [PIC32MM](#) is the sole 32-bit part that doesn't use an ARM core.

AVR

The AVR core is a famous RISC design known for its clock-cycle efficiency – especially at the time it was introduced in 1997. I reviewed two microcontrollers with an AVR core – the [tinyAVR 1-Series](#) and the [megaAVR](#).

The specific AVR instruction set and timing for both parts I reviewed is known as “AVRe” – this instruction set includes a two-cycle multiply and many single-cycle operations. Note that tinyAVR parts prior to the [tinyAVR 1-Series](#) are essentially completely different MCUs with a less-capable AVR core that has no multiplier.

The AVR core has a 16-bit instruction fetch width; most instructions are 16 bits wide; some are 32. Still, this is a RISC architecture, so the instruction set is anything but orthogonal; while there are 32 registers you can operate with, there are very few instructions for working directly with RAM; and of those 32 registers, I'd say that only 16 of them are true "general purpose" registers, as R0-R15 can't be used with all register operations (load-immediate probably being the most important).

All things said, though, AVR offered a huge performance improvement over the 12-cycle or 6-cycle 8051 processors when AVR was first introduced – and the AVR is always faster than even modern 8051 derivatives when it comes to working with large arrays of data that must be stored in extended (16-bit) RAM on the 8051.

It was also designed for C compilers, too – with 32 registers available at all times, compilers can efficiently juggle around many operands concurrently; the 8051, by comparison, has four banks of eight registers that are only easily switched between within interrupt contexts (which is actually quite useful).

And interrupts are one of the weak points of the AVR core: there's only one interrupt priority, and depending on the ISR, many registers will have to be pushed to the stack and restored upon exit. In my testing, this often added 10 PUSH instructions or more – each taking 2 cycles.

Another issue with AVR is the generally slow clock speed – even the high-end XMEGA AVR parts can only run at up to 32 MHz, with both the parts reviewed here topping out at 20 MHz. Compare that to the [EFM8](#), of which is many varieties run at 48 MHz or higher (like the 72 MHz Laser Bee I reviewed). Even a 50%-better clock cycle efficiency doesn't help much when the competition runs almost four times faster than the AVR.

MICROCHIP PIC16

There's something fundamentally goofy about almost all aspects of the [PIC16](#) that make it seem, at first glance, completely bizarre that it is as popular as it is.

PIC16 uses an odd-ball 14-bit-wide program memory, yet it's an 8-bit machine. This dramatically simplifies the core architecture: a 14-bit word can hold just enough data to specify every CPU instruction – with enough free space left in the word to address up to 128 registers or 2K of program memory (for the two jump/call routines).

Microchip calls the PIC16 a RISC machine since every PIC instruction (there's just 49 of them) is precisely one word long. There's considerable debate as to the precise definition of a "RISC architecture" is, but while the PIC16 has a single-word instruction length for all instructions, the PIC16 varies greatly from most RISC parts in that it is an accumulator-based machine, and has no working registers. I'll leave it up to you to decide. The PIC16 is often described as a 4T architecture – taking 4 clock cycles to execute a single machine instruction. This isn't *entirely* true, as the PIC16 takes an additional 4 cycles to *fetch* that instruction. Consequently, it's actually an 8T machine, though it implements a pipeline scheme that allows each instruction to execute in 4 cycles – except jumps, which take 8.

Since real MCUs have more than 128 bytes of registers and 2K of program memory, this PIC has a bank selection register (BSR), which is written to whenever you need to swap banks (which happens *a lot*).

The PIC16 is a single-register machine, and that register is named *W*. Everything you do will essentially be moving *something* into *W*, doing *something* with it, and then moving it back to *somewhere*. Consequently, programming it in assembly is easy, and downright fun.

Because this part can store 8192 14-bit program words, Microchip will tell you this part has 14 KB of flash (close to 16 KB, right?), but users will tell you that it has 8K of program memory – 8192 *words* of memory – since storing an 8192-element byte array will occupy all 14 KB of its flash memory. Keep this in mind when comparing memory.

I have a longer write-up of PIC16 in the [main PIC16 article](#).

MICROCHIP PIC24

While the PIC10, 12, 16, and 18 are all 8-bit cores with 12-16 bit program memory, the PIC24 moves up to 16-bit data operated through 24-bit instructions (are you starting to catch onto the numbering system?)

While all the PICs before it were 4T machines, the [PIC24](#) is 2T – that is, two clock cycles per instruction cycle.

The PIC24 has new indirect addressing modes that allow incrementing/decrementing and register-offset addressing, has a few more other instructions, and has three – instead of two – hardware breakpoints; but otherwise, the core is very much in the spirit of the [PIC16](#).

The [PIC24](#) carries the excellent power consumption figures that the [PIC16](#) has, but many of the parts lack the clocking and oscillator options the [MSP430](#) has (and apples-to-apples, the MSP430 is lower-power).

The dsPIC versions of these parts – which add DSP-friendly instructions – are popular for motor drivers, but it’s not clear that the PIC24 has more widely been the runaway success Microchip had hoped.

MICROCHIP PIC32

While everyone was migrating their 8-bit proprietary cores to Arm, Microchip was gleefully popping out PIC parts. But in 2007, they finally decided to add a new microcontroller – the [PIC32](#) – which uses a third-party, industry-standard 32-bit core. Instead of following everyone to the Arm ecosystem, they took a different turn: PIC32 parts use the MIPS architecture – specifically the M4K core.

MIPS built this core for single-chip MCU applications. M4K has 32 registers, a 5-stage pipeline, vectored interrupts and exceptions, bit-manipulation, and 16-bit instruction encoding support.

It is not the same as an Arm processor, but at the C application level, they are similar enough that any Arm developer should have no problems (other than the usual manufacturer-to-manufacturer peripheral differences).

You can program and debug the PIC32 using the same PGC/PGD/MCLR set-up you use on all other PIC parts – but there’s also support for JTAG, though the most popular JTAG debugger – the Segger J-Link – has limited support for the PIC32MX parts, and no support for the PIC32MM.

ARM CORTEX-M0

The Arm Cortex-M0³ Formerly ARM, but as of August 1, 2017, “Arm” is the capitalization style [they now use](#). is a 32-bit RISC architecture that serves as the entry-level Arm architecture available to silicon vendors for microcontroller applications. Arm cores are designed by [Arm](#)

[Holdings](#) and licensed to semiconductor manufacturers for integration into their products.

Arm started out as a personal computer microprocessor when Advanced RISC Machines formed a joint venture between Acorn, Apple, and VLSI Technology to manufacture 32-bit processors for the Acorn computer. While Arm cores have grown in popularity as microprocessors for battery-powered systems (they are almost certainly powering your smartphone), Arm moved into the microcontroller sphere as well – the ARM7TDMI-S was probably the first Arm core that was used in microcontrollers – i.e., processors with completely self-contained RAM, flash, and peripherals. The Atmel AT91 and ST STR7 were probably the first microcontroller parts designed with an Arm core.

It's important to understand the history of Arm because it explains a serious feature of Arm microcontrollers that differs substantially from the 8051 (the other multi-vendor architecture that dominates the field): Unlike the 8051, Arm is just a core, not a complete microcontroller.

The ARM7TDMI-S didn't come with any GPIO designs, or provisions for UARTs or ADCs or timers – it was designed as a microprocessor. Thus, as vendors started stuffing this core into their extremely high-end MCUs, they had to add in their vendor-specific peripherals to the AHB (AMBA⁴ Advanced Microcontroller Bus Architecture – these multi-level acronyms are getting tedious High-performance Bus).

Consequently, Freescale used a lot of HC08 and ColdFire peripherals; while Atmel designed new peripherals from scratch. ST borrowed a bit from the ST7 (the precursor to the [STM8](#)) but used new designs for timers and communications peripherals.

Since many microcontroller projects spend 90% or more of the code base manipulating peripherals, this is a serious consideration when switching from one Arm MCU vendor to another: there's absolutely zero peripheral compatibility between vendors, and even within a single vendor, their Arm parts can have wildly different peripherals.

Unlike other Arm parts, the M0 series only supports a subset of the 16-bit Thumb instruction set, which allows it to be about 1/3 the size of a Cortex-M3 core. Still, there's a full 32-bit ALU, with a 32-bit hardware multiplier supporting a 32-bit result. Arm provides the option of either a single-cycle multiply, or a 32-cycle multiply instruction, but in my browsing, it seems as though most vendors use the single-cycle multiply option.

In addition to the normal CPU registers, Arm cores have 13 general-purpose working registers, [which is roughly the sweet spot](#). The core has a nested vector interrupt controller, with up to 32 interrupt vectors and 4 interrupt priorities – plenty when compared to the 8-bit competition, but a far cry from the 240 interrupts at 256 interrupt priorities that the larger Arm parts support. The core also has full support for runtime exceptions, which isn't a feature found on 8-bit architectures.

The M0+ is an improved version of the M0 that supports faster two-cycle branches (due to the pipeline going from three-stage to two-stage), and lower power consumption. There are a slew of silicon options that vendors can choose from: single-cycle GPIO, support for a simple

instruction trace buffer called Micro Trace Buffer (MTB), vector table relocation, and a rudimentary memory protection unit (MPU).

One of the biggest problems with ARM microcontrollers is their low code density for anything other than 16- and 32-bit math – even those that use the 16-bit Thumb instruction set. This means normal microcontroller type routines – shoving bytes out a communication port, wiggling bits around, performing software ADC conversions, and updating timers – can take a lot of code space on these parts. Exacerbating this problem is the peripherals, which tend to be more complex – I mean “flexible” – than 8-bit parts, often necessitating run-time peripheral libraries and tons of register manipulation.

Another problem with ARM processors is the severe 12-cycle interrupt latency. When coupled with the large number of registers that are saved and restored in the prologue and epilogue of the ISR handlers, these cycles start to add up. ISR latency is one area where a 16 MHz 8-bit part can easily beat a 72 MHz 32-bit Arm microcontroller.

8051

The 8051 was originally an Intel microcontroller introduced in 1980 as one of the first widely-deployed 8-bit parts. The 8-bit modified Harvard core has a fully-orthogonal variable-length CISC instruction set, hardware multiplier and hardware divider, bit-addressable RAM and specific bit-manipulation instructions, four switchable banks of eight registers each, two-priority interrupt controller with automatic register bank-switching, 64 KB of both program and extended RAM addressability, with 128 bytes of “scratch pad” RAM accessible with fast instructions.

The 8051 was actually a specific *part* – not a family – but its name is now synonymous with the core architecture, peripherals, and even package pin-out⁵ the 8051 is a member of a family officially called the “MCS-51” – along with the 8031, 8032, 8051, and 8052 – plus all the subsequent versions that were introduced later.

The original had 4K of ROM⁶ Expensive windows ceramic packages allowed [EPROM programming](#) for developers, but production units were mask-ROM or OTP – or, in the case of the 8031, only external ROM was supported., 128 bytes of RAM, four full 8-bit GPIO ports (32 I/O total), a UART, two or three timers, and a two-priority interrupt system.

The 8051 has a fully orthogonal CISC instruction set, which means you can do nearly any operation with immediate, direct, or indirect operands, and you can do these operations in RAM, registers, or the A accumulator.

Many vendors built direct, drop-in compatible clones of the 8051, and by the time Intel discontinued its MCS-51 products in 2007, barely anyone noticed – these third-party parts evolved from mimicking Intel functionality to outright beating them with tons of additional timers, peripherals, and special-purpose functionality.

Because of its small core and fast interrupt architecture, the 8051 architecture is extremely popular for managing peripherals used in real-time high-bandwidth systems, such as USB web cameras and audio DSPs, and is commonly deployed as a house-keeping processor in FPGAs used in audio/video processing and DSP work.

Old-timers associate the 8051 with “old and slow” because the original was a 12T microcontroller – each machine cycle took 12 clock cycles to complete. Since there was no pipelining, every instruction byte took a machine cycle to fetch, plus one or more additional machine cycles to execute – altogether, it could take more than 50 clock cycles to execute a given instruction. Ouch.

No need to worry about that anymore, though: all the modern 8051-style MCUs available are 1T processors, and many of them have a pipelined core, meaning many instructions take the same number of clock cycles to execute as the instruction’s length. Consequently, in many 8051 implementations, operations in A (the accumulator) are the fastest, followed by the register bank, and then RAM.

I ended up with three different 8051-compatible microcontrollers end the lineup: the [Nuvoton N76](#), the [Silicon Labs EFM8 Laser Bee](#), and the [STCmicro STC8](#).

From a pure core design standpoint, the [STC8](#) is probably the most interesting – while I can’t find documentation to confirm this, it appears that STCmicro uses a 24-bit (or more) parallel instruction-fetch size. This means that a huge number – more than 80% – of instructions can be executed in a single cycle, which is a substantial step up from all other 8051s on the market these days.

However, the [EFM8](#), from Silicon Labs’ C8051 lineage, can hit much higher clock speeds – topping out at 72 MHz – making it the fastest part I reviewed, by core speed.

I’ll be looking more at this in the performance section of the review.

STM8

The **STM8** core has six CPU registers: a single accumulator, two index registers, a 24-bit program counter, a 16-bit stack pointer, and a condition register. The STM8 has a Harvard architecture, but uses a unified address space. There's a 32-bit-wide program memory bus which can fetch most instructions in a single cycle – and pipelined fetch/decode/execute operations permit many instructions to execute in a single cycle.

The claim to fame of the core is its comprehensive list of 20 addressing modes, including indexed indirect addressing and stack-pointer-relative modes. There's three “reaches” for addressing – short (one-byte), long (two-byte), and extended (three-byte) – trading off memory area with performance.

This is the only architecture in this round-up that has this level of granularity – all the other chips are either RISC-style processors that have lots of general-purpose registers they do their work in, or 8051-style CISC parts that manipulate RAM directly – but pay a severe penalty when hitting 16-bit address space. The STM8 manages these trade-offs in an efficient manner.

PERIPHERALS

Use the tabs below to compare precise specs across families.

SPEED	FLASH	RAM	TIMERS	PWM	COMMS	ADC
ATMEL TINYAVR						20 MHZ
ATMEL MEGA AVR						20 MHZ

ATMEL SAM D10	48 MHZ
CYPRESS PSOC 4000S	24 MHZ
FREESCALE KE04	48 MHZ
FREESCALE KL03	48 MHZ
HOLTEK HT66	20 MHZ
INFINEON XMC1100	32 MHZ
MICROCHIP PIC16	32 MHZ
MICROCHIP PIC24	32 MHZ
MICROCHIP PIC32MM	25 MHZ
NUVOTON N76	16 MHZ
NUVOTON M051	50 MHZ
NXP LPC811	30 MHZ
RENESAS RL78	24 MHZ
SANYO LC87	12 MHZ
SILICON LABS EFM8	72 MHZ

ST STM8**16 MHZ****ST STM32F0****48 MHZ****STC STC8****30 MHZ****TI MSP430FR****16 MHZ**

The chart above illustrates the differences in core clock speed among each MCU. As will be seen in the evaluation section, core clock speed is not a good predictor of performance when comparing between different MCU families (especially between 8-, 16-, and 32-bit parts). However, most MCUs limit the maximum peripheral clock rate to that of the CPU, which may be a driving factor if your application requires fast peripheral clocks (say, for fast GPIO bit-banging or for high-speed capture/compare timer operations). The Infineon XMC1100 is a neat exception to this rule – its peripheral clock can run at up to 64 MHz.

There are other important asterisks to this data: the Atmel tinyAVR and megaAVR parts have severely limited operating ranges when running below 5V, which will affect most modern designs. The tinyAVR can only run at 10 MHz below 3.6V, and at 5 MHz below 2.2V. The megaAVR has the same speed grades, but even worse, has nothing faster than an 8 MHz internal oscillator. When talking about sub-\$1 MCUs, adding a crystal or even low-cost ceramic resonator adds a sizable portion of the cost of the MCU to the BOM.

The Silicon Labs EFM8 Laser Bee, with its 72 MHz core clock speed, beats out even the ARM microcontrollers in this round-up. The Sanyo LC87 brings in a 12 MHz reading – but bear in mind this is a 3T architecture, which limits the actual instruction clock speed to 4 MHz. The Holtek HT66 and Microchip PIC16 are both 4T architectures, but the PIC16 has a relatively snappy 32 MHz core speed (thanks to its on-board PLL), which allows it to compete better with 8 MHz parts.

ATMEL MEGAAVR	2.1 X
ATMEL SAM D10	4.5 X
CYPRESS PSOC 4000S	9.2 X
FREESCALE KE04	16.4 X
FREESCALE KL03	14.7 X
HOLTEK HT66	1.0 X
INFINEON XMC1100	23.6 X
MICROCHIP PIC16	8.0 X
MICROCHIP PIC24	19.4 X
MICROCHIP PIC32MM	66.0 X
NUVOTON N76	3.2 X
NUVOTON M051	10.2 X
NXP LPC811	27.0 X
RENESAS RL78	10.5 X
SANYO LC87	4.2 X

SILICON LABS EFM8

3.9 X

ST STM8

2.6 X

ST STM32F0

27.0 X

STC STC8

1.0 X

TI MSP430FR

15.3 X

PARAMETRIC REACH

One of the major themes of this microcontroller selection guide is to show how easy it can be to get going with different parts, and comfortably jump around among ecosystems – picking the best part for the job.

But for casual hobbyists who may live far away from major distributors, and professionals who have to meet tight timelines, sometimes there's no time to play around with new architectures.

If you want to commit to a single architecture, it's important to know which one gives you the most headroom to move up. I created a fictitious “times better” score by comparing the the part tested with the best part available in the same ecosystem – this usually means fairly comparable peripheral programming, along with identical development tools. I multiplied the core speed, package size, flash, and RAM capacities together, ratioed the two parts, and then took the quartic root. Essentially, if every parameter is double, it is considered “2.0 x” as powerful.

Surprisingly, the [PIC32](#) came out on top: there are four PIC32 families – the PIC32MX was the first; it's the mainstream core that runs up to 120 MHz, with up to 512 KB of flash and 128 KB of RAM.

But it's the newer PIC32MZ that reaches even higher: up to 252 MHz, with 2 MB of flash, and – with the DA version of the part – includes 32 MB of onboard DDR2 memory. With the MZ-DA, you can build complex graphical apps without needing an application processor running

Linux (and the PCB / BSP complexity that arrives with that). It's essentially the PIC32 version of the Arm Cortex-M7.

Next up, the [STM32](#) line. The STM32F0 has a famous big brother – the STM32F4 – that's one of the most capable Arm Cortex parts ever built. Several versions run up to 180 MHz, with 2 MB of flash and up to 364 KB of RAM (in the case of the STM32F469).

But the brand-new STM32F7 – part of the new Cortex-M7 line of parts – goes even further, with 216 MHz maximum operating frequency, 2 MB of flash, and 512 KB of RAM.

The [LPC811](#) – one of the lower-performing parts in my round-up – has several big sisters, including the LPC546xx series, a giant Cortex-M4 with 220 MHz max frequency, 512 KB of flash, 200 KB of RAM, in up to 208-pin packages.

The [tinyAVR](#) in this review has very little headroom – these devices top out at 16 KB of flash, 2 KB of RAM, 20 MHz, and 24-pin QFN packages; however, 32 KB tinyAVR parts are soon to be released.

The [megaAVR](#) has a bit of reach in pinning and memory. The ATmega3290 keeps the 20 MHz clock speed but bumps up the pin-count to 100 pins. There are many megaAVR parts with 64 KB of flash and 4 K of RAM, as this part has. Some megaAVR parts have as much as 16 KB of RAM or 256 KB of flash. Oddly, Atmel can't seem to combine these specs – there is no 100-pin, 256 KB flash, 16 KB RAM megaAVR that is in current production.

The [SAM D10](#) extends up to the SAM D21, which maintains its 48 MHz clock speed, but increases flash up to 256 KB, with up to 32 K of RAM, and sizable 64-pin package options.

There's a lot of headroom left in the SAM Arm microcontroller ecosystem, but if you leave the D1x/D2x line of parts, you'll lose familiarity with some of the peripherals (especially the communications interfaces). Having said that, the Arm Cortex-M7-based ATSAMS70 will get you up to 300 MHz of performance, 2 MB of flash, and 384 KB of RAM, in up to 144-pin packages. The older A revision parts are deeply discounted, including this 64-pin 1 MB flash part for \$5.14 on DigiKey.

With the [PSoC 4000S](#), you're at rock-bottom in the PSoC ecosystem, so the only direction is up – the PSoC 5 devices run at 80 MHz, with up to 256 KB of flash and 64 KB of RAM in 100-pin packages. These parts come with all the PSoC goodness that Cypress users love – reconfigurable digital logic, lots of analog features, and excellent capacitive-touch sensing.

The just-around-the-corner PSoC 6 promises to bring even more performance to the ecosystem, with the PSoC 63 running a 150 MHz Arm Cortex-M4F, 1 MB of flash, 288 KB of RAM, integrated BTLE connectivity, and 100+ pin packages.

The Freescale [KE04](#) and [KL03](#) are both entry-level devices in the E and L families within the Kinetis system. The E family has good reach up to the 168 MHz KE1x, with up to 512 KB of flash, 64 KB of RAM, and up to 100-pin packaging. Unfortunately, this is a fairly different process than the KE04 reviewed here – there’s no Processor Expert support, and the communications peripherals are quite different than those in the lower-end part. It retains its 5V operating range, which segments it into the E family.

The KL03 extends up to the KL28 – a 72 MHz Cortex-M0+ with 512 KB of flash, 128 KB of RAM, and up to 121-pin BGA packages available. Unlike the E series, the KL series has much more uniform peripherals across its range of devices.

The [Holtek HT-66](#) has no real latitude above the HT66F0185 – but as mentioned on the review page, there are tons of application-specific products Holtek makes that use this core.

The Infineon [XMC1000](#) family extends up to the XMC1400, with 200 KB of flash, 16 K of RAM, a slightly-faster 48 MHz core clock, and a 64-pin package. Moving out of the XMC1000 ecosystem, the XMC4000 keeps the XMC1000 peripherals and swaps out the core for a Cortex-M4F, running up to 144 MHz, with 2 MB of flash, 352 KB of RAM, and up to 196-pin packaging options.

The [PIC16](#) tops out in the 64-pin PIC16F19197 device, with 56 KB of flash (well, 32 K words, because PIC), 4 K of RAM. I’ll include PIC18 devices, though, as they’re targetted with the same compiler, programmed with the same debugger, and share peripherals and architectural decisions.

PIC18 devices can reach up to the PIC18F97J60 – a 100-pin beast with 128 KB of flash (64 K words), and almost 4K of RAM. While most of these 8-bit parts have similar peripherals across the board, I must note the Ethernet MAC and PHY present in the PIC18F97J60. While many higher-end microcontrollers have an Ethernet MAC, this low-end PIC18 part is one of the only microcontrollers – at any price – to also integrate a PHY. The only other mainstream MCU that has an integrated Ethernet PHY is the \$14 Tiva-C TM4C129x, a giant 128-pin 120 MHz Arm Cortex-M4 from Texas Instruments. There are a few other (albeit odd) choices out there: Freescale’s legacy ColdFire microcontrollers include the MCF5223X, which has an integrated Ethernet PHY. Fabless designer ASIX manufactures the AX11015, a 100 MHz 8051 with an integrated Ethernet PHY.

The PIC24FJ1024GA610 supports up to a 1024 KB of flash, 32 KB of RAM, and a 32 MHz run speed – though there are dsPIC devices, like the dsPIC33EP512GM604, with up to 140 MHz operating frequency, 512 KB of flash, and 48 KB of RAM.

The [Nuvoton N76](#) can look up to the 40 MHz W79E658A, with 128 KB of flash, 100-pin packaging, and 1.25K of RAM.

The [Nuvoton M051](#) has a few Cortex-M4 big sisters: the M505 has 2 MB of embedded SPI flash (sounds slow to me), and 128 KB of RAM, runs at 100 MHz, and comes in a 48-pin package. Nuvoton hasn’t released the M487 yet, but it promises a 192 MHz CPU with 512 KB of flash, 160 KB of RAM, and at least 80-pin package options (though Nuvoton hasn’t unveiled all details yet).

[Renesas RL-78](#) extends up to the R5F101SLAFB, with 128-pin package, 512 KB of flash, 32 KB of RAM, and a slightly-faster 32 MHz clock speed.

Sanyo's LC87 extend up to 100-pin 256 KB parts with 12 KB of RAM, and an 18 MHz clock with the LC87F7NP6AUE.

The [Silicon Labs EFM8LB1](#) is the top-of-the-line part in the EFM8 family – these top out at 64 KB of flash, 4.25 KB of RAM, and a relatively-small 32-pin package.

Having said that, you can stay in the Silicon Labs 8051 family with the C8051F120 – a 100 MHz 8051 with 128 KB of flash, 8.25 KB of RAM, and a 100-pin microcontroller. It's an older part, but it's still supported in Simplicity Studio (though not in Simplicity Configurator).

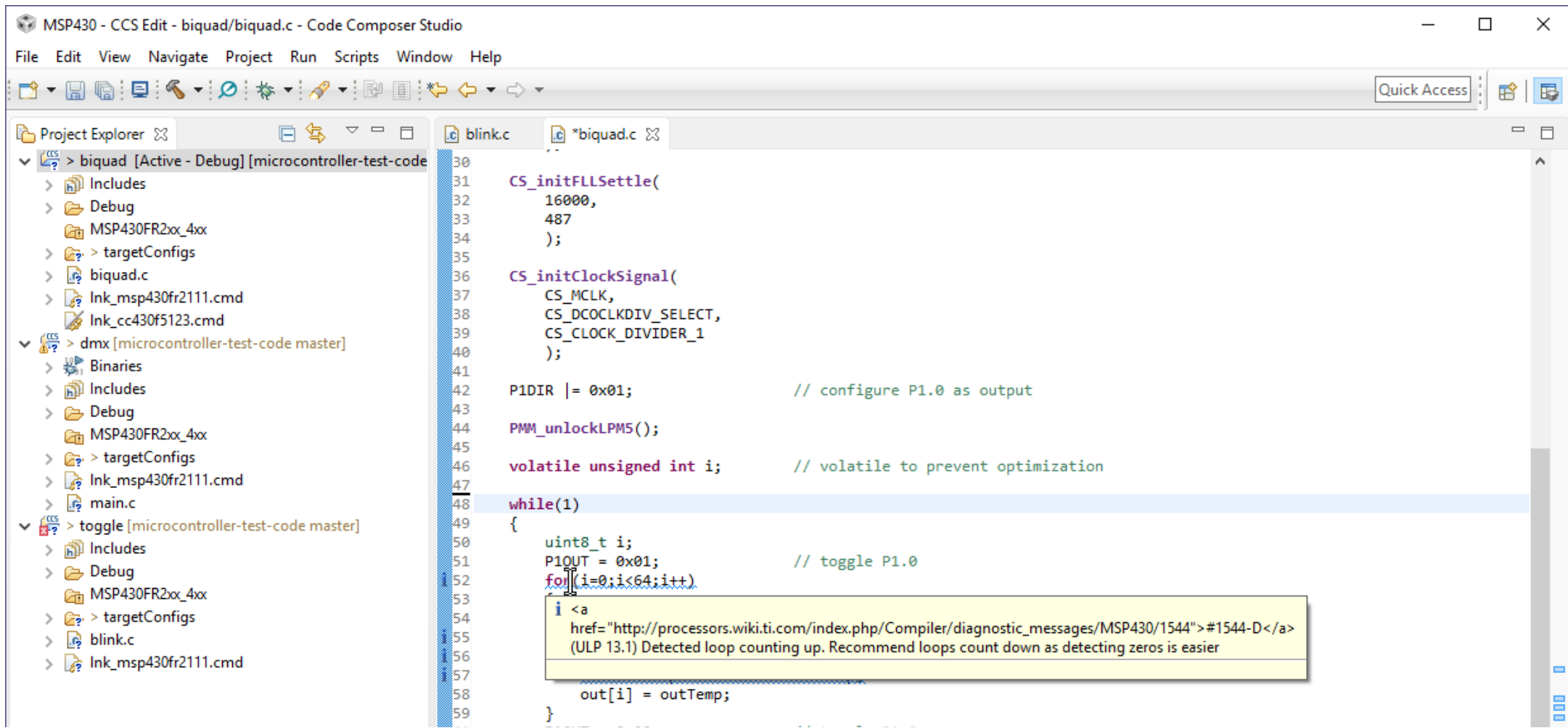
The [STC8](#) I tested is the top-of-the-line part in their catalog.

The [MSP430](#) extends up to the MSP430F6779, with 512 KB of flash, 32 KB of RAM, and 128-pin packaging.

DEVELOPMENT
ECOSYSTEM

The development ecosystem of a microcontroller has a profound impact on productivity and ease of use of the part, and these IDEs, peripheral libraries, dev boards, and debuggers varied wildly among the microcontrollers reviewed here.

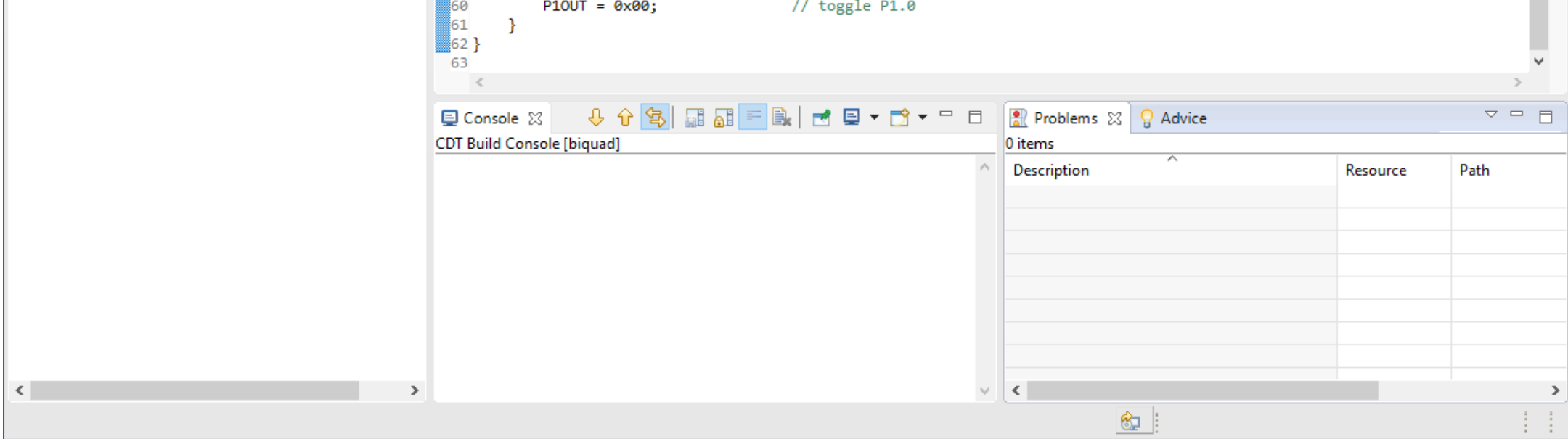
DEVELOPMENT ENVIRONMENTS



The screenshot shows the Code Composer Studio IDE for MSP430. The Project Explorer on the left shows a project named 'biquad' with subfolders for 'Includes', 'Debug', 'MSP430FR2xx_4xx', 'targetConfigs', and 'biquad.c'. The main editor window displays the file 'blink.c' with the following code:

```
30
31 CS_initFLLSettle(
32     16000,
33     487
34 );
35
36 CS_initClockSignal(
37     CS_MCLK,
38     CS_DCOCLKDIV_SELECT,
39     CS_CLOCK_DIVIDER_1
40 );
41
42 P1DIR |= 0x01;           // configure P1.0 as output
43
44 PMM_unlockLPM5();
45
46 volatile unsigned int i; // volatile to prevent optimization
47
48 while(1)
49 {
50     uint8_t i;
51     P1OUT = 0x01;        // toggle P1.0
52     for(i=0;i<64;i++)
53     {
54         i <a
55         href="http://processors.wiki.ti.com/index.php/Compiler/diagnostic_messages/MSP430/1544">#1544-D</a>
56         (ULP 13.1) Detected loop counting up. Recommend loops count down as detecting zeros is easier
57     }
58     out[i] = outTemp;
59 }
```

A yellow tooltip is visible over the code, displaying the error message: (ULP 13.1) Detected loop counting up. Recommend loops count down as detecting zeros is easier.



Eclipse-based development environments, such as Code Composer Studio from Texas Instruments, provide a complete text editor, toolchain, and debugging system in one application — plus many vendors choose to extend Eclipse with vendor-specific features, such as the ULP Advisor that Texas Instruments bundles to help developers get active- and sleep-mode current down to the minimum.

ECLIPSE

Eclipse is a Java-based IDE originally developed at IBM to develop in Java. But since 2001, it has been an open-source project built by the Eclipse Foundation.

[Eclipse CDT](#) provides C/C++ tooling, and has taken off like wildfire in the embedded world — starting in the 32-bit ARM ecosystems, but migrating down to 16- and 8-bit parts as well. In fact, almost all the major microcontrollers here are programmed in an IDE based on Eclipse:

[NXP Kinetis KE04](#) uses [Kinetis Design Studio](#)

[NXP Kinetis KL03](#) uses [MCUXpresso](#) (or [Kinetis Design Studio](#))

[Infineon XMC1100](#) uses [DAVE](#)

Nuvoton M051 uses [CooCox CoIDE](#)

NXP LPC811 uses [MCUXpresso](#)

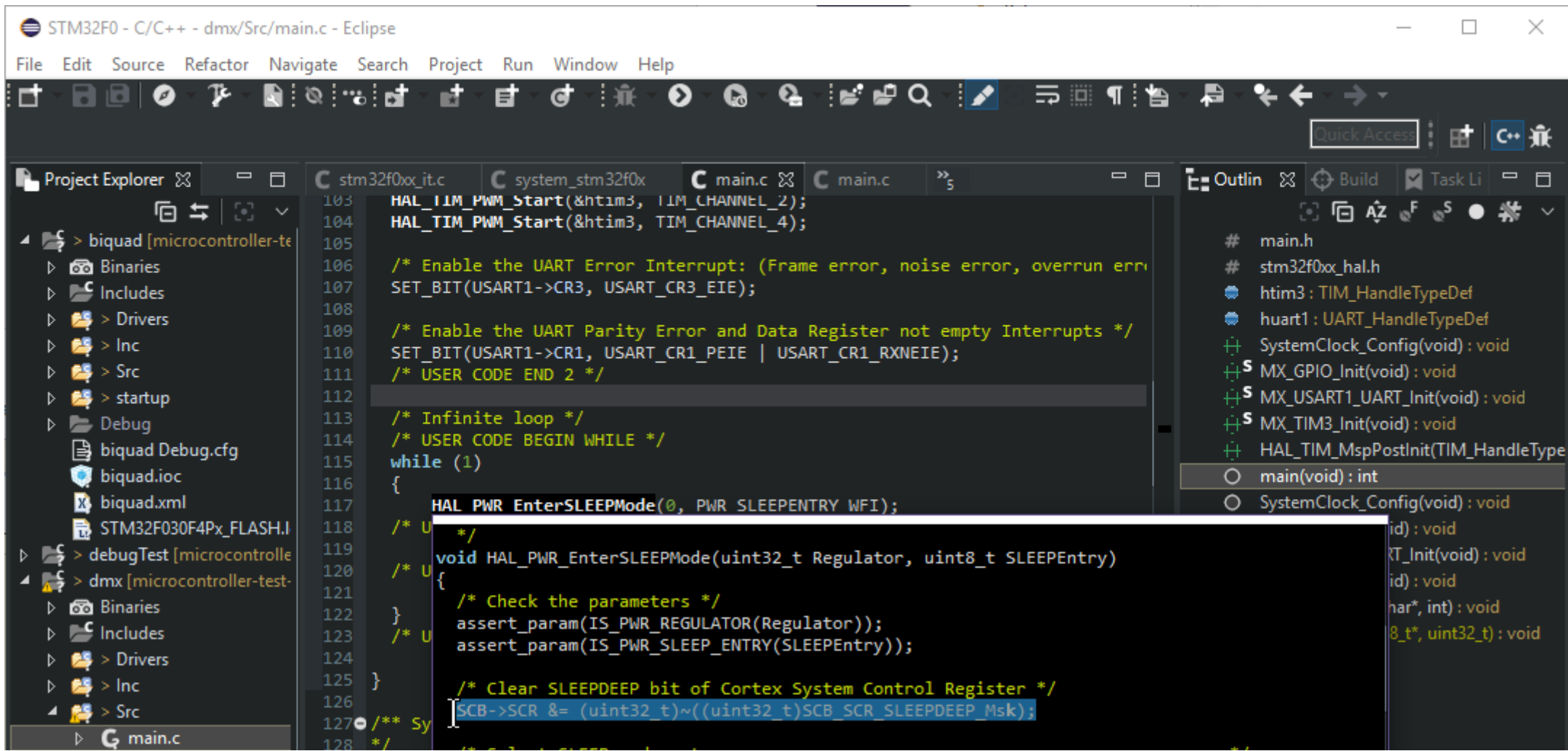
Renesas RL-78 uses [e² studio](#)

Silicon Labs EFM8 uses [Simplicity Studio](#)

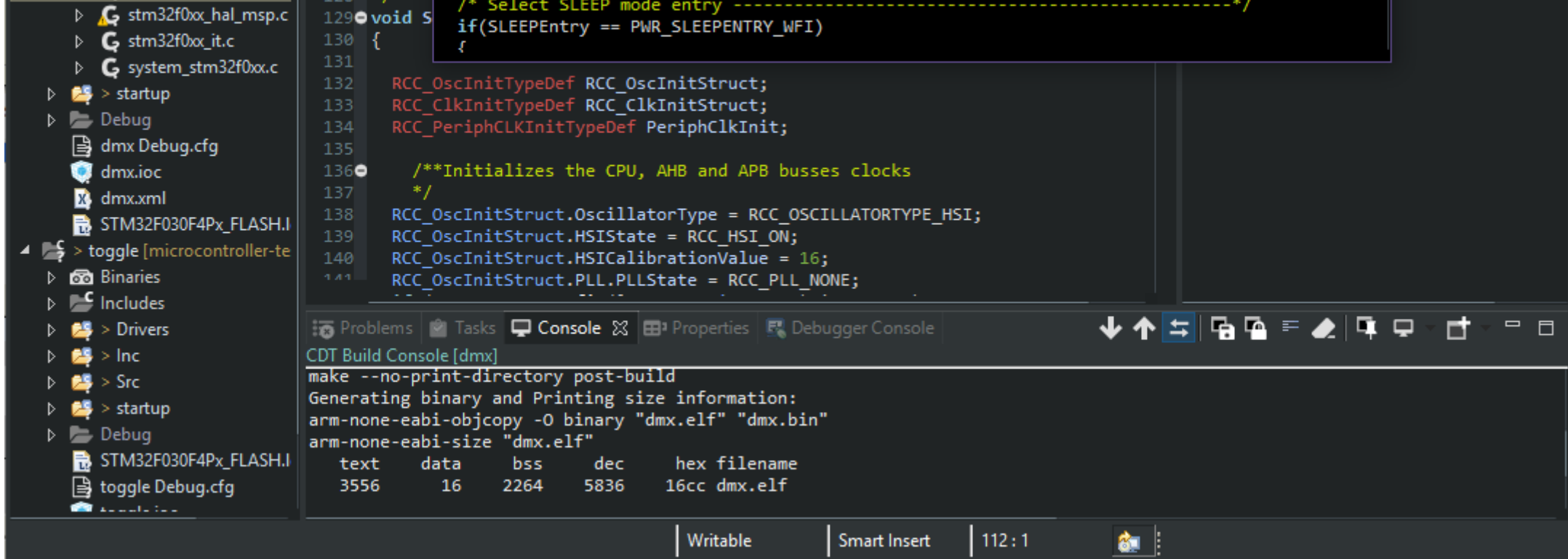
ST STM32F0 uses [System Workbench for STM32](#)

Texas Instruments MSP430 uses [Code Composer Studio](#)

Other than DAVE, CooCox, and e² studio (which all only run on Windows), all of these toolchains have cross-platform support for Windows, macOS, and Linux.



```
STM32F0 - C/C++ - dmx Src/main.c - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Project Explorer
  biquad [microcontroller-test]
    Binaries
    Includes
    Drivers
    Inc
    Src
    startup
    Debug
    biquad Debug.cfg
    biquad.ioc
    biquad.xml
    STM32F030F4Px_FLASH.I
  debugTest [microcontroller-test]
  dmx [microcontroller-test]
    Binaries
    Includes
    Drivers
    Inc
    Src
      main.c
      stm32f0xx_it.c
      system_stm32f0xx.c
      main.c
      main.c
  Outlin
  Build
  Task List
  # main.h
  # stm32f0xx_hal.h
  htim3 : TIM_HandleTypeDef
  huart1 : UART_HandleTypeDef
  SystemClock_Config(void) : void
  MX_GPIO_Init(void) : void
  MX_USART1_UART_Init(void) : void
  MX_TIM3_Init(void) : void
  HAL_TIM_MspPostInit(TIM_HandleTypeDef) : void
  main(void) : int
  SystemClock_Config(void) : void
  id) : void
  T_Init(void) : void
  id) : void
  nar*, int) : void
  8_t*, uint32_t) : void
103 HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_2);
104 HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_4);
105
106 /* Enable the UART Error Interrupt: (Frame error, noise error, overrun error) */
107 SET_BIT(USART1->CR3, USART_CR3_EIE);
108
109 /* Enable the UART Parity Error and Data Register not empty Interrupts */
110 SET_BIT(USART1->CR1, USART_CR1_PEIE | USART_CR1_RXNEIE);
111 /* USER CODE END 2 */
112
113 /* Infinite loop */
114 /* USER CODE BEGIN WHILE */
115 while (1)
116 {
117     HAL_PWR_EnterSLEEPMode(0, PWR_SLEEPENTRY_WFI);
118     /* USER CODE BEGIN WHILE */
119     void HAL_PWR_EnterSLEEPMode(uint32_t Regulator, uint8_t SLEEPEntry)
120     {
121         /* Check the parameters */
122         assert_param(IS_PWR_REGULATOR(Regulator));
123         assert_param(IS_PWR_SLEEP_ENTRY(SLEEPEntry));
124     }
125     /* Clear SLEEPDEEP bit of Cortex System Control Register */
126     SCB->SCR &= (uint32_t)~((uint32_t)SCB_SCR_SLEEPDEEP_Msk);
127     /* USER CODE END WHILE */
128 }
```



Recent versions of Eclipse, like the 4.6.3 Neon release that System Workbench for STM32 uses, provide good dark theme support, as well as the incredibly snappy pop-up function browser that shows you the full source code of any function you hover over.

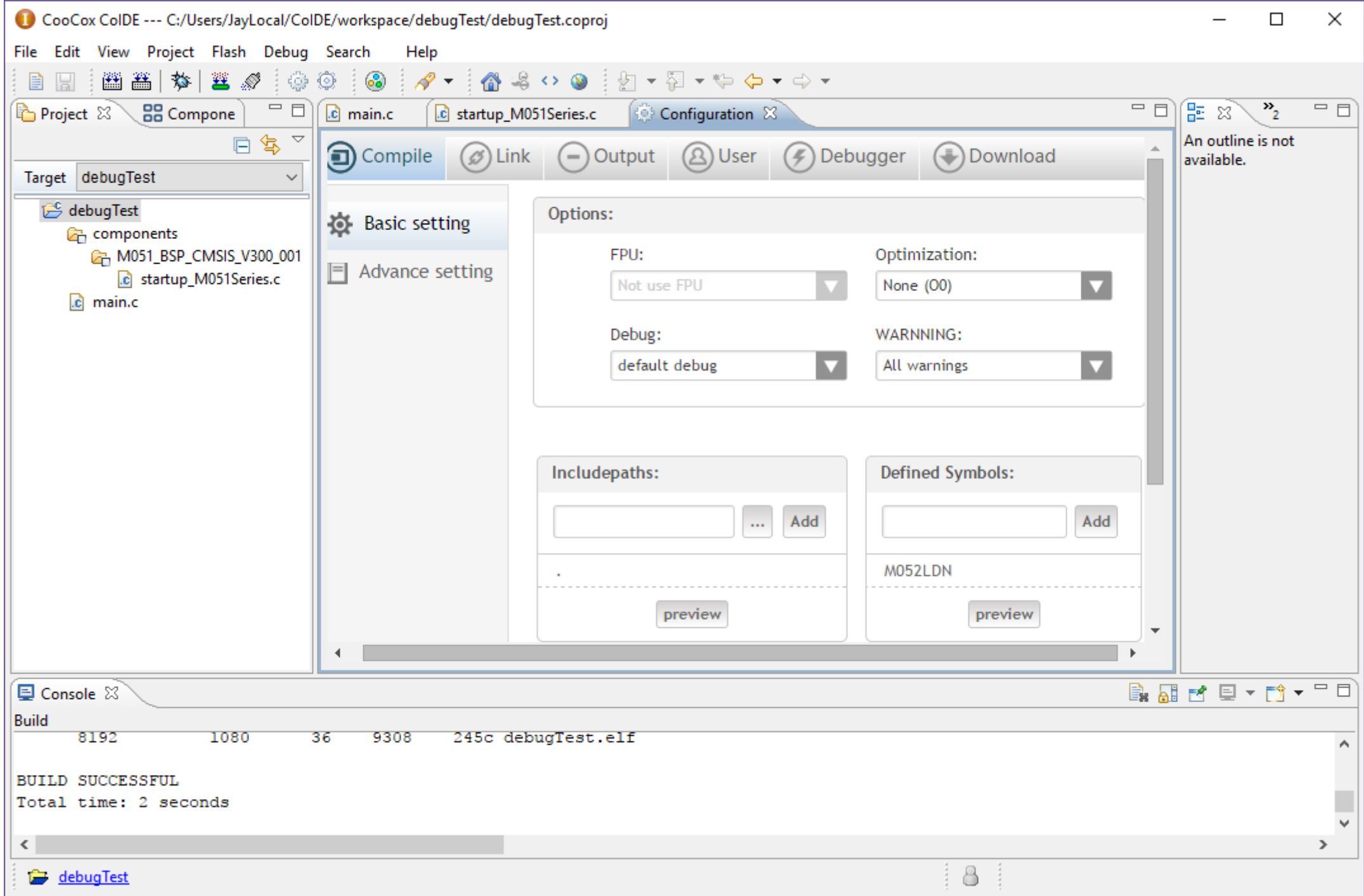
Main features of Eclipse CDT include multi-project workspaces, debugging with tons of introspection windows, support for different toolchains, plus a syntax-highlighting text editor with content assist text completion, macro definition browsing and highlighting, code snippets, and tons of refactoring capabilities.

The code editor in Eclipse is definitely a stand-out among IDEs tested – especially if you’re coming from other IDEs. Everything is completely customizable, very snappy, and full of features. The pop-up “Source Hover” is one of my favorite features: hover over a function, and the source of the function (including any docs) will pop-up immediately. If you’re still not sure about something, move your mouse down into the pop-up window and it turns into a scrollable editor window, allowing you to see the entire contents of the function (and copy-and-paste from it). One feature request: I would *love* to see Ctrl-Click working from *within* this pop-up, and it would also be amazing to see editing capability, too.

VENDOR CUSTOMIZATIONS

The [GNU MCU Eclipse plug-ins](#) have made it trivial to set up an Eclipse-based workflow when working on many ARM processors; under the hood, some of these IDEs are basically just pre-packaged open-source components with a nice splash screen. This is definitely the case for Kinetis Design Studio, MCUXpresso, and System Workbench for STM32 (which didn't even bother changing the Eclipse logo). There's nothing wrong with that – it's far less jarring to move between stock Eclipse IDEs, and if you're short on hard drive space, you could probably install and configure plug-ins to essentially combine many of these IDEs together.

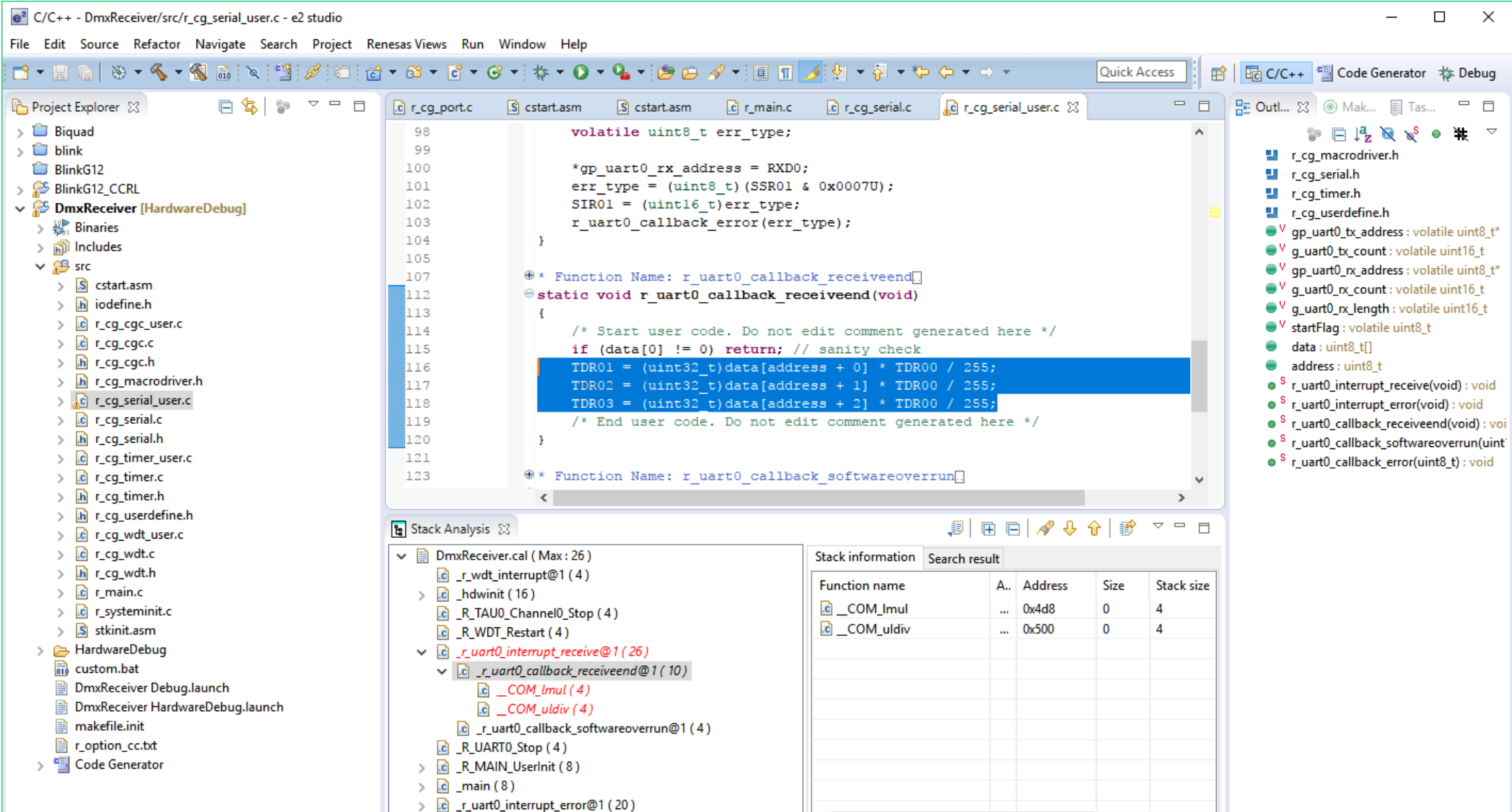
On the other hand, some vendors had to go through great lengths to get an Eclipse-based environment working with their parts. Silicon Labs had to write a debug interface from scratch that could communicate with their tools (and work with Keil C51 binaries), custom property panes for managing the build system – along with packaging a patched WINE system that can run Keil C51 seamlessly on macOS and Linux (and from my testing, they pulled it off).

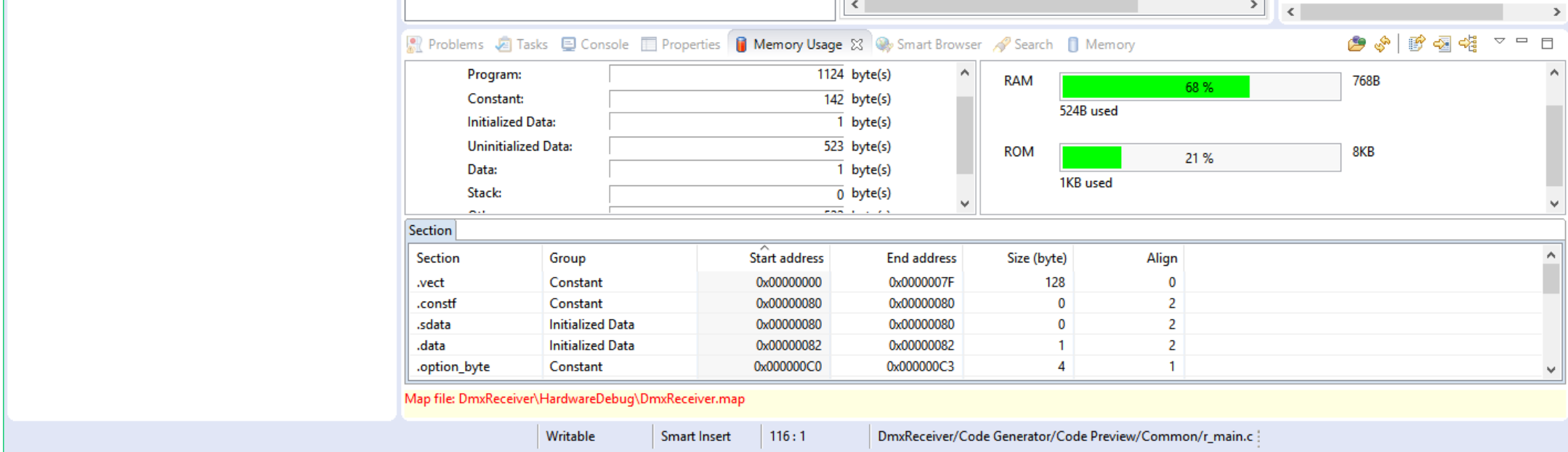


CoIDE has a super-simple target configuration property pane, and Keil μ Vision-style buttons for building and rebuilding projects.

In fact, one IDE – CoIDE – is so far removed from Eclipse, I hesitated to even mention it in this list. CoCox essentially stripped Eclipse down to its base, and built up CoIDE saving little more than the Eclipse shell.

What results is an IDE that is extremely easy to use – great for students and hobbyists who many find the Eclipse project properties pane to be... well, a pain. This comes at the expense of flexibility, however – tons of the debugging options and windows are missing, and you can only open one project at a time. Sometimes the IDE does thing you may not want. For example, every folder you create in a project is automatically added to the list of project source files *as well as include files*. I imagine this will quickly cause filename conflicts in large projects (which CoIDE doesn't seem built for).



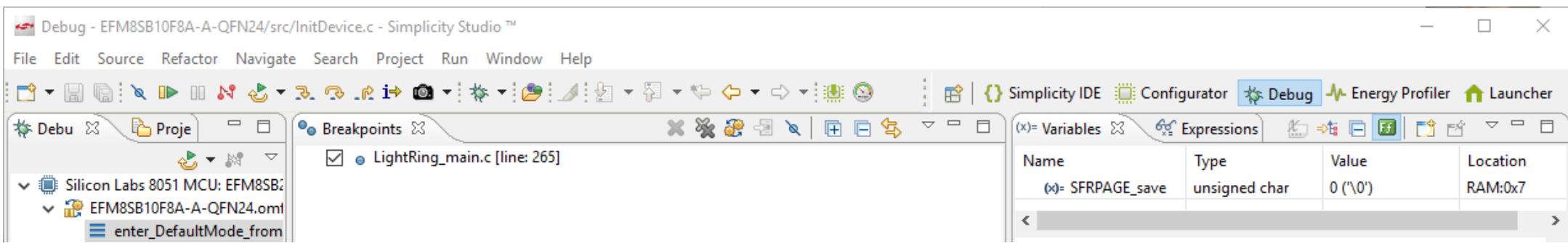


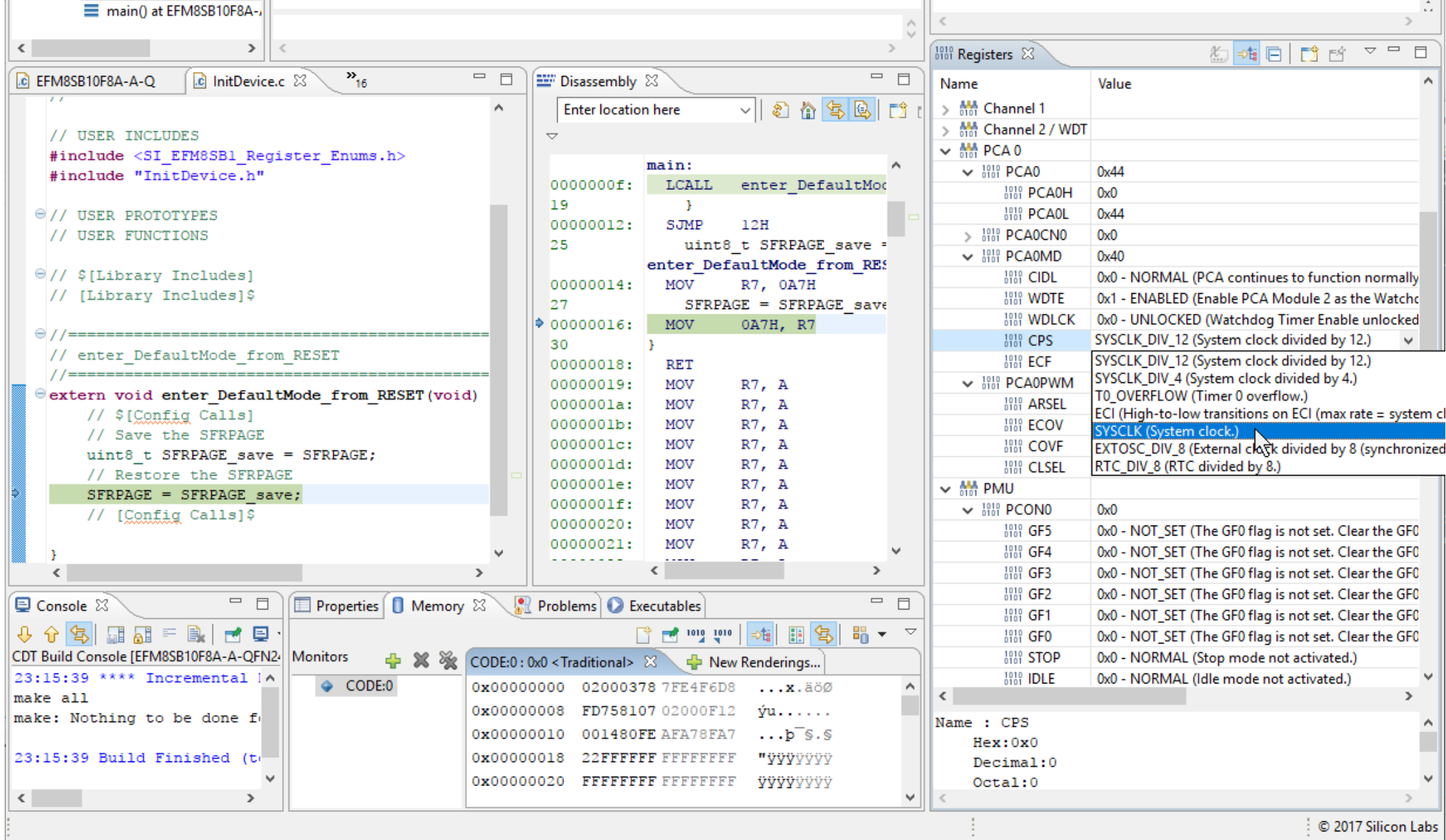
When building out the DMX-512 demo for the RL-78, e² studio's Stack Analysis tool helped me see where my stack-heavy calls were.

Vendors like Freescale decided to stop heavily customizing Eclipse with their own proprietary debugging system (which they did in [CodeWarrior](#)), and switch to these open-source plug-ins. I have mixed feelings about this change, as CodeWarrior seemed much faster at starting and stopping debug sessions than the GDB-based system everyone uses these days.

On performance overall, recent versions of Eclipse (Mars or better) seem to be much snappier than the [disastrous Juno release](#), so if it felt slow and bloated last time you tried it, you may want to give it another shot.

Oxygen (4.7.0) was just released in June, so the newest IDEs – System Workbench, MCUXpresso, and Code Composer Studio – and are still on Neon (4.5). Other Eclipse-based IDEs are on older versions – with Kinetis Design Studio and DAVE being on the oldest release (Luna SR2 – 4.4.2).





Simplicity Studio has a beautiful and functional register view that allows you to interact with registers using names and drop-down lists — this saves a lot of time when tracking down problems — no datasheet required.

DEBUGGING IN ECLIPSE

Eclipse – across nearly all vendors – provided the best out-of-the-box experience out of all the IDEs I tested. Projects support multiple debug configurations that allow you to use different debuggers and target configurations. Out of the box, you get a source code view with interactive breakpoints, a memory browser, a disassembly view, and a list of core CPU registers.

Debugging in Eclipse is relatively unified across the platforms – the biggest differences are the supported debuggers, and the custom debugging windows – especially the peripheral register view.

In my opinion, the peripheral register viewer is one of the most important debug windows – even more important than the disassembly view. My favorite register viewers are the ones used in TI’s Code Composer Studio and SiLabs’ Simplicity Studio. These have a contiguous list of all registers, organized in a tree view, with detailed descriptions of each register’s value, with combo-box selectors of all the values. Simplicity Studio edges out Code Composer Studio by naming all the constants. This is sometimes unnecessary (like: SYSCLK_DIV_12 (System clock divided by 12)), but definitely keeps your eyes out of the datasheet for the part.

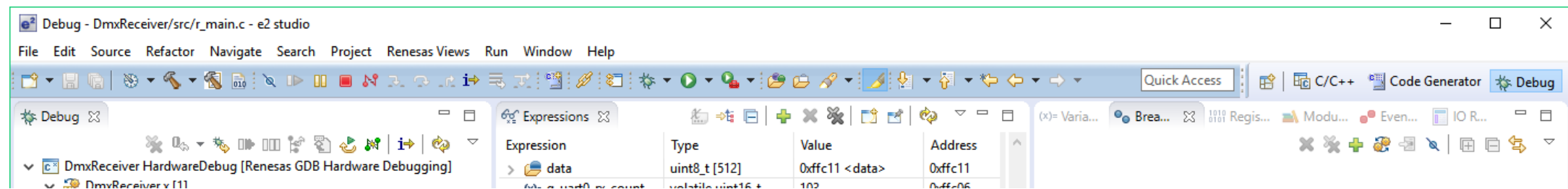
Kinetis Design Studio uses [EmbSysRegView](#) – an open-source Eclipse plugin – which provides similar functionality. This uses [SVD](#), a standardized set of tools for generating header files and descriptions of such.

System Workbench for STM32 has a similar view, but doesn’t automatically fetch the register values. This may not be a bad thing, as ARM microcontrollers tend to have a lot more register addresses than 8-bit parts, but it still feels a bit clunkier.

MCUXpresso and Infineon DAVE have a somewhat-strange two-step process – you select the peripheral in one view, and it creates a memory rendering. Registers are broken out logically into one or more bits, but the drop-down lists don’t have named enumerations that describe what each bit-pattern does.

DAVE has exactly the same plugin as MCUXpresso, but it seems buggy – it doesn’t always work. Hopefully this gets fixed in a future version, because it severely limits the usability of the debug system.

CoIDE and e² studio both have the worst register views – they simply display a list of the peripheral register whole values, without breaking them up logically or annotating them with text.



Thread #1 1 (single core) (Running)
 C:/Renesas/e2_studio/DebugComp/r178-elf-gdb (7.8.2)
 GDB server

(x) g_uart0_rx_count	volatile uint16_t	165	0xffc08
(x) g_uart0_rx_length	volatile uint16_t	512	0xffc0a
R red	uint16_t	0	0xffc0c
R green	uint16_t	165	0xffc0e
R blue	uint16_t	66	
+ Add new expression			

```

cstart.asm | r_main.c | r_cg_serial_user.c
67     while (1U)
68     {
69 000004ba     if(startFlag)
70             {
71 000004bf         startFlag = 0;
72 000004c2         R_UART0_Receive(data, 512);
73             }
74 000004cc     HALT();
75     }
76     /* End user code. Do not edit comment generated here */
77 }
78
79 /* Function Name: R_MAIN_UserInit
80 void R_MAIN_UserInit(void)
81 {
82     /* Start user code. Do not edit comment generated here */
83     EI();
84     R_UART0_Start();
85     R_TAU0_Channel0_Start();
86     /* End user code. Do not edit comment generated here */
87 }
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2
```

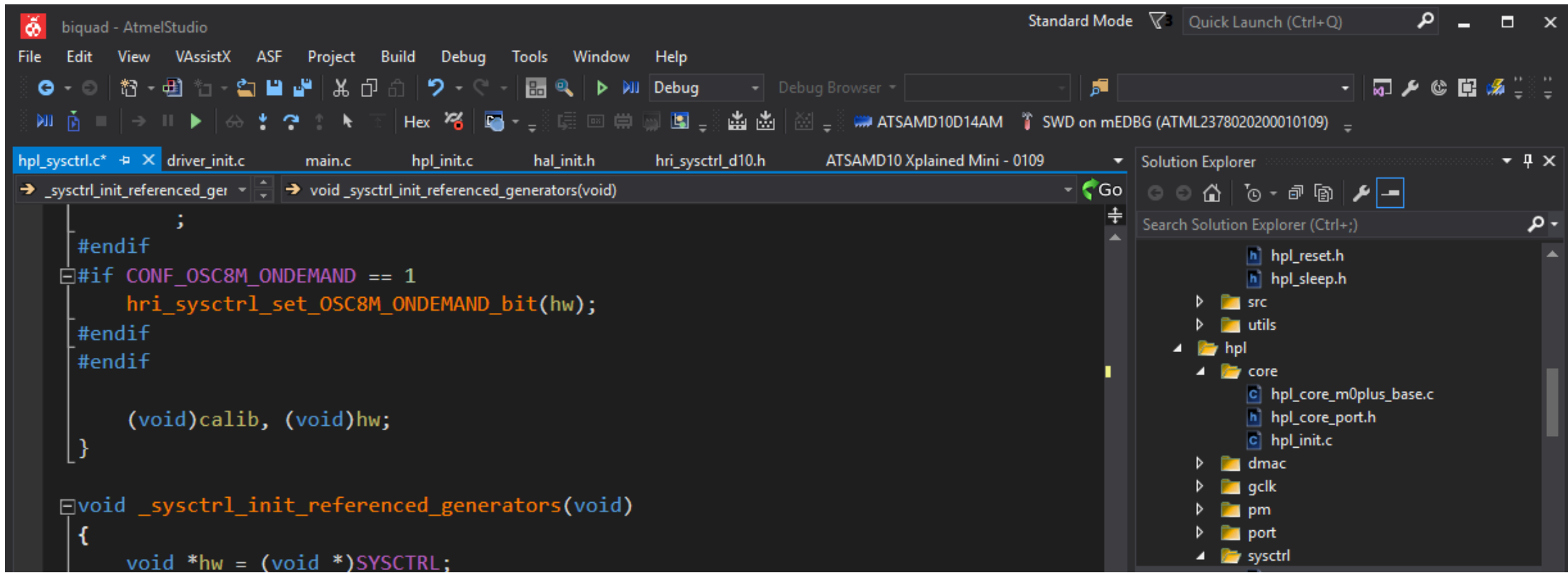
OTHER FEATURES

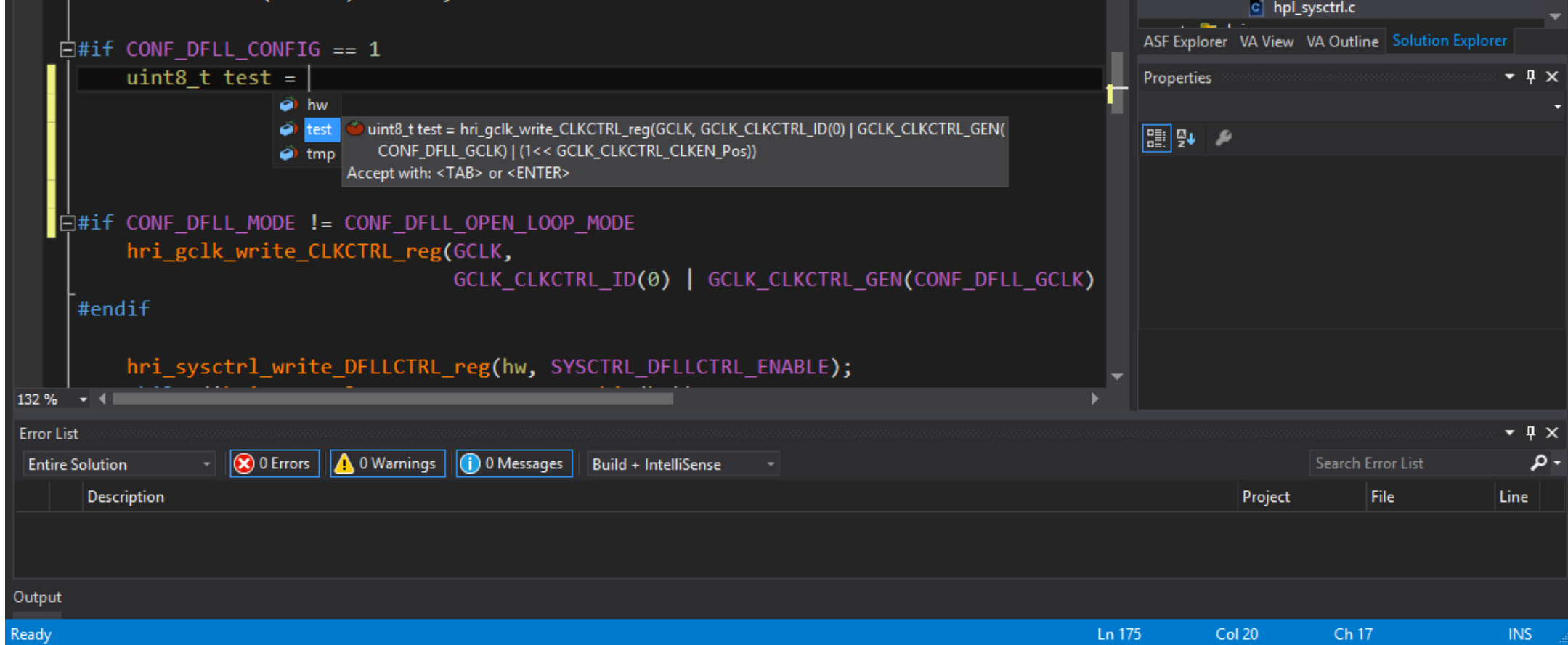
Some vendors have gone above and beyond with useful (and not-so-useful) additional views and features. e² studio, despite the pesky peripheral register viewer, has a useful real-time view of variables that update while debugging. I suspect the IDE is periodically breaking the MCU, reading the contents of RAM, and updating the display – but they may have a mechanism for real-time tracing in the RL-78 core.

This IDE can also show flash and RAM usage – though it's not plotted as nicely as it is in DAVE, which shows a pie chart of all symbols.

Specific to ARM parts, all Eclipse IDEs I tested support semihosting, which allows you to print characters to a console window during debugging. The characters are printed through the debugging interface, so there's no need to configure a UART. MCUXpresso and CoIDE were the easiest Eclipse-based IDEs to configure with semihosting.

MCUXpresso has nice project properties panes for selecting a C runtime library (it includes Redlib in addition to Newlib Nano) as well as linker settings.





Which of these #ifdefs are enabled? Your guess is as good as mine; Atmel Studio is Visual Studio without Microsoft's excellent IntelliSense engine, making it worse than even Keil μ Vision in terms of text-editing productivity — and far inferior to the Eclipse- and NetBeans-based IDEs from competitors. I added 6 publicly-visible global variables in this file among others in the project, and none of them appear in the auto-complete list.

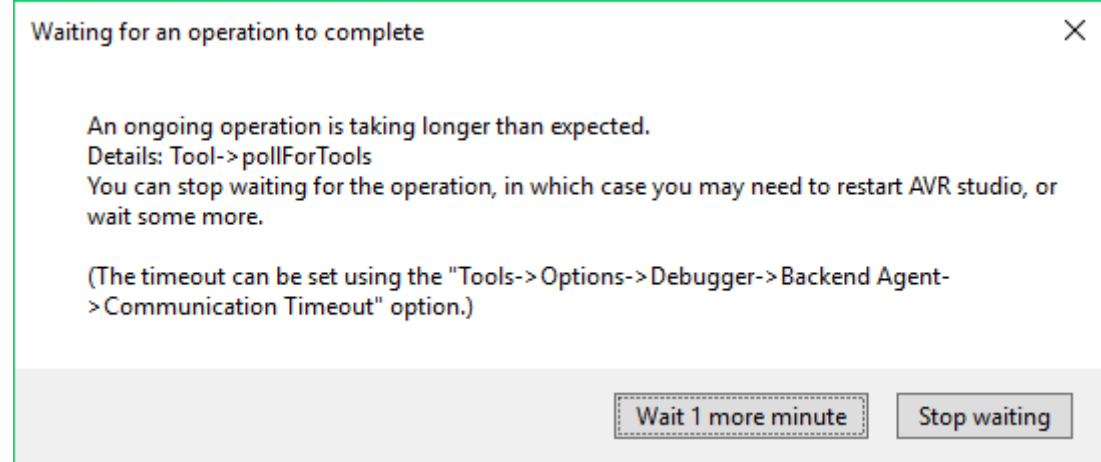
ATMEL STUDIO

While many vendors have transitioned to Eclipse-based IDEs, Atmel went with a Visual Studio Isolated Shell-based platform starting with AVR Studio 5. I do a ton of .NET and desktop-based C++ development, so I expected to feel right at home in Atmel Studio when I first launched it. Unfortunately, Microsoft calls this product "Visual Studio Isolated Shell" for a reason — it's simply the *shell* of Visual Studio, without any of the

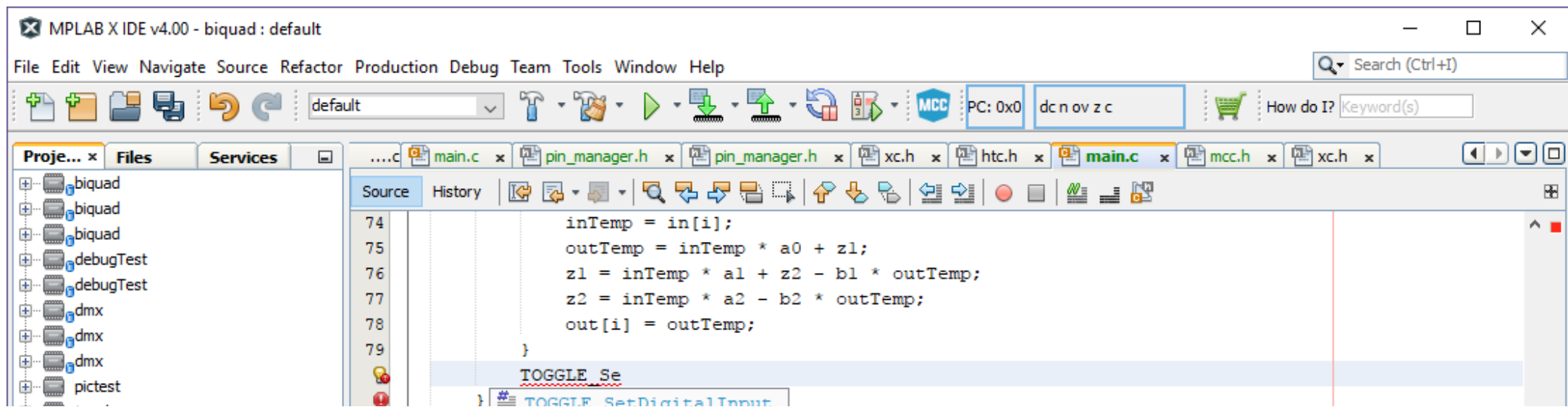
meat. The excellent IntelliSense engine that Microsoft spent years perfecting has been replaced by a third-party “Visual Assist” plugin that struggles to identify global variables, evaluate pre-processor definitions, or perform refactoring of items defined outside of the current file. The Toolchain editor is a near-clone of the Eclipse CDT one (no reason to reinvent the wheel), but it’s missing checkboxes and inputs for commonly-used compiler and linker options; one stunning omission is link-time optimization, which even when manually-specified as command parameters, doesn’t seem to work – odd, since Atmel is using a recent version of GCC.

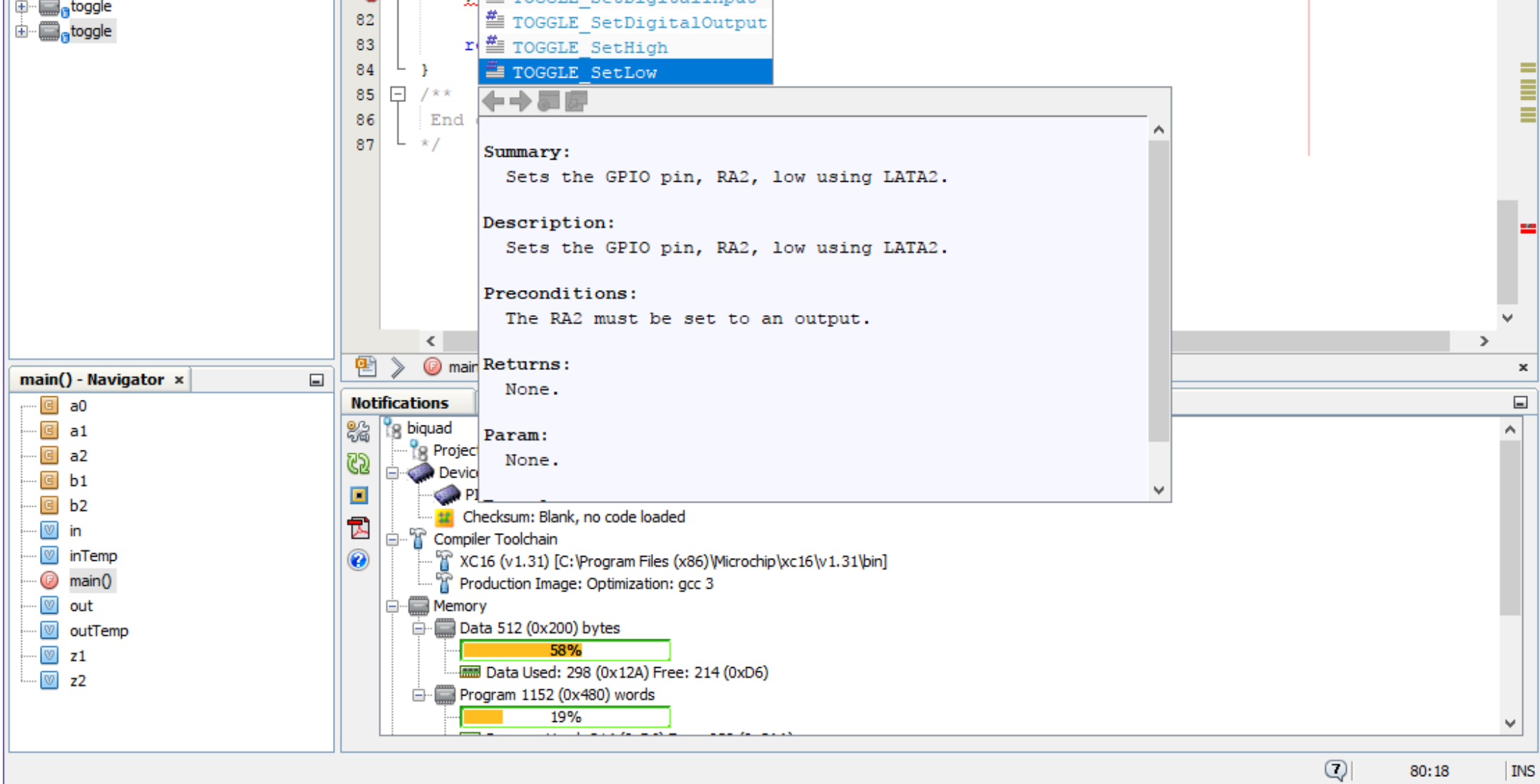
My biggest issue with Atmel Studio is how incredibly buggy and unstable it has been every time I’ve used it in the last two years. I’m not referring to a specific installation on a specific computer:

rather, every single time I’ve installed the software, I’ve fought with AVR Dragon drivers, a bad DLL file in the installer, programmer firmware issues, or, most recently, the software popping up the “Waiting for an operation to complete” message that prevents me from debugging *any* Atmel product without restarting my computer. Look, I get it: embedded firmware development is a highly-specialized task, so maintaining software that works reliably for such a small user-base can be challenging. Yet, every other vendor tools tested worked nearly flawlessly.



This error started popping up recently in Atmel Studio — the only solution seems to restart my computer. It’s obviously from an old chunk of code, since it’s referring to the program as “AVR Studio.”





MPLAB X is a NetBeans-based IDE that is comparable to Eclipse in terms of OS support, editor capabilities, and debugging — but it lacks some of the advanced debug configurations and introspection features that make Eclipse so powerful.

MPLAB X

While many vendors were moving from their proprietary Windows-only IDEs to an open-source Eclipse-based workflow, Microchip went a different route when they moved MPLAB 8 to the NetBeans-based MPLAB X in 2012.

On paper, NetBeans is a lot like Eclipse – it’s a Java-based IDE that was originally built to target Java development – but has since expanded to support C/C++ (along with web-centric languages: HTML, PHP, JavaScript, etc). Like Eclipse, NetBeans is open-source, and cross-platform. Unlike Workspaces in Eclipse, NetBeans doesn’t strongly enforce this paradigm – but it does offer “Project Groups” which has similar functionality. Both have good text-completion capabilities and source introspection; both have macro expansion.

I’ve used both for years, so I feel comfortable making this subjective claim: NetBeans feels simpler; Eclipse feels more powerful. A lot of this is a result of the UX design choices – Eclipse loads the window with tons of buttons, drop-down menus, and docked panes full of features. The entire IDE’s scale is much more dense than NetBeans.

Even though NetBeans has a lot of the same features, the UI is sparsely populated with the bare minimum of buttons you need to get your job done. Even the menu bar is light on options. Instead, advanced, rarely-used features are buried away inside sub-sub menus, or – somewhat more commonly – with keyboard combinations. As an example, I have no idea how to show the excellent Macro Expansion view in NetBeans, other than pressing Ctrl-Alt and clicking on a macro. Just to double-check, I went hunting for it in the menu bar, as well as digging through the context menus.

I think students and hobbyists might be drawn toward the simplicity of NetBeans, but I prefer Eclipse’s density, as it encourages users to go exploring and discover new features.

One big omission with the NetBeans text editor is the pop-up Source Hover code explorer that Eclipse has. Hover over any function in Eclipse, and the entire source code for that function pops up in a window you can scroll through. NetBeans will display code docs for functions, but if you want to look at the content of them, you’ll have to Ctrl-Click your way into the definition.

MICROCHIP CUSTOMIZATIONS

I have to applaud Microchip for heavily customizing NetBeans into MPLAB X – an IDE that really feels like it was built for embedded development. The Project Properties window is all Microchip – you can select which tool you want to use to program the device (or the integrated simulator), as well as compiler options, include paths, and tool configuration.

Integrating the tool setup into individual project properties is useful for developers who switch between devices (and voltages!) a lot; but it may feel clunky to users who are always on the same device, using the same settings – every time they create a new project, they’ll have to go through the same tool configuration settings (debugger powers target, select correct voltage, blah blah blah).

One goof immediately visible is the redundant “Run Project” and “Make and Program Device” buttons. From what I can tell, these are identical (the manual says you can use either). However, from the bizarre Eclipse integrations I’ve seen, I’m used to dealing with UI oddities like this when dealing with embedded IDEs.

Microchip integrates a lovely dashboard view (visible in the bottom of the main photo), which indicates the device target, the compiler, and the memory usage.

The MPLAB X Options window allows you to add multiple instances of the Microchip XC compilers. As these compilers are [quite expensive](#) and don’t support new devices introduced after their release, it is common for shops to have several versions of XC compilers floating around – old, paid-for versions, and new code-size-limited versions they may grab to evaluate a new part before forking over dough.

MPLAB X IDE v4.00 - dmx : default

File Edit View Navigate Source Refactor Production Debug Team Tools Window Help

Search (Ctrl+I)

default

How do I? Keyword(s)

Files Ser...

main.c x main.c x main.c x main.c x interrupt_manager.c x mcc.c x main.c x mcc.h...

Source History

```

65     LATA2 = 1;
66     if(U1STAbits.FERR == 1)
67     {
68         index = 0;
69     }
70
71     // Receive Data Ready
72     // there is a 4 byte hardware Rx fifo, so we must be sure to get all read bytes
73     while (U1STAbits.URXDA)
74     {

```

Notifications Output dmx - Dashboard SFRs x Variables Call Stack Breakpoints

Address	Name	Hex	Decimal	Binary	Char						
220	U1MODE	0xC000	49152	11000000 00000000	'À.'						
222	U1STA	0x8510	34064	10000101 00010000	'Q.'						
224	U1TXREG	0x0000	0	00000000 00000000	'...'						
226	U1RXREG	[U1STA]	15	14	13	11	10	9	8		
228	U1BRG			UTXISEL1	UTXINV	UTXISEL0	-	UTXBRK	UTXEN	UTXBF	TRMT
2C0	TRISA		1	0	0	-	0	1	0	1	
2C2	PORTA										
2C4	LATA		6	5	4	3	2	1	0		
2C6	ODCA				URXISEL	ADDEN	RIDL	PERR	FERR	OERR	URXDA
2C8	TRISB		00	0	1	0	0	0	0		
2CA	PORTB	0x0200	512	00000010	00000000	'..'					
2CC	LATB	0x0000	0	00000000	00000000	'..'					
2CE	ODCB	0x0000	0	00000000	00000000	'..'					
2FC	PADCFG1	0x0000	0	00000000	00000000	'..'					
4E0	ANSA	0x000F	15	00000000	00001111	'...'					
4E2	ANSB	0xC010	49168	11000000	00010000	'À.'					

Memory SFRs Format Individual

dmx (Build, Load, ...) debugger halted 107:1 INS

I'm not a big fan of the hover-over-to-view-bit-values feature in the peripheral registers view.

DEBUGGING

Debugging across all Microchip devices is much slower than in other IDEs. I'm not sure if this a limitation of the MCU, the PicKit 3, or the IDE (or all of the above). The default behavior of MPLAB X is to reconnect from the tool whenever starting a debug session, but you can shave a few seconds off the debug load time by instructing the IDE to maintain a constant connection to the tool by ticking the appropriate box in the Options dialogue. This really ought to be the default option, as few developers know of its existence, debugging is slow enough as it is, and there are very few usage cases where you'd want the IDE to disconnect from the debugger upon completing a debug session.

While there's an option to display a Disassembly view while debugging, this doesn't come up by default, and even when you select it, it doesn't seem to be "sticky" – you have to re-open the view every time you start a debug session. This is fine for Java or desktop C/C++ development, but for embedded microcontrollers, disassembly view is critical.

Another afterthought seems to be the peripheral register view, and peripheral register handling in general. Pull up the SFRs view, and you'll be greeted by an extremely slow-loading window that's painful to scroll through – especially on larger devices like the PIC32.

This view displays the address of the register (who cares?), the hard-to-understand short-form name, the hex value of the register, the decimal value of the register (who cares?), the binary value (hilariously long when working on 16- and 32-bit processors), and an ASCII interpretation of the register's value (...why on Earth...?) – but what's *not* in this view is: a human-readable name of the register, description of the register's function, or a human-readable decoding of the register's current value.

There's a hover-over break-down of the register into its individual bits, but these use the same short-form datasheet names, don't provide descriptions, and don't contain enumeration values for multi-bit fields (or any explanation at all).

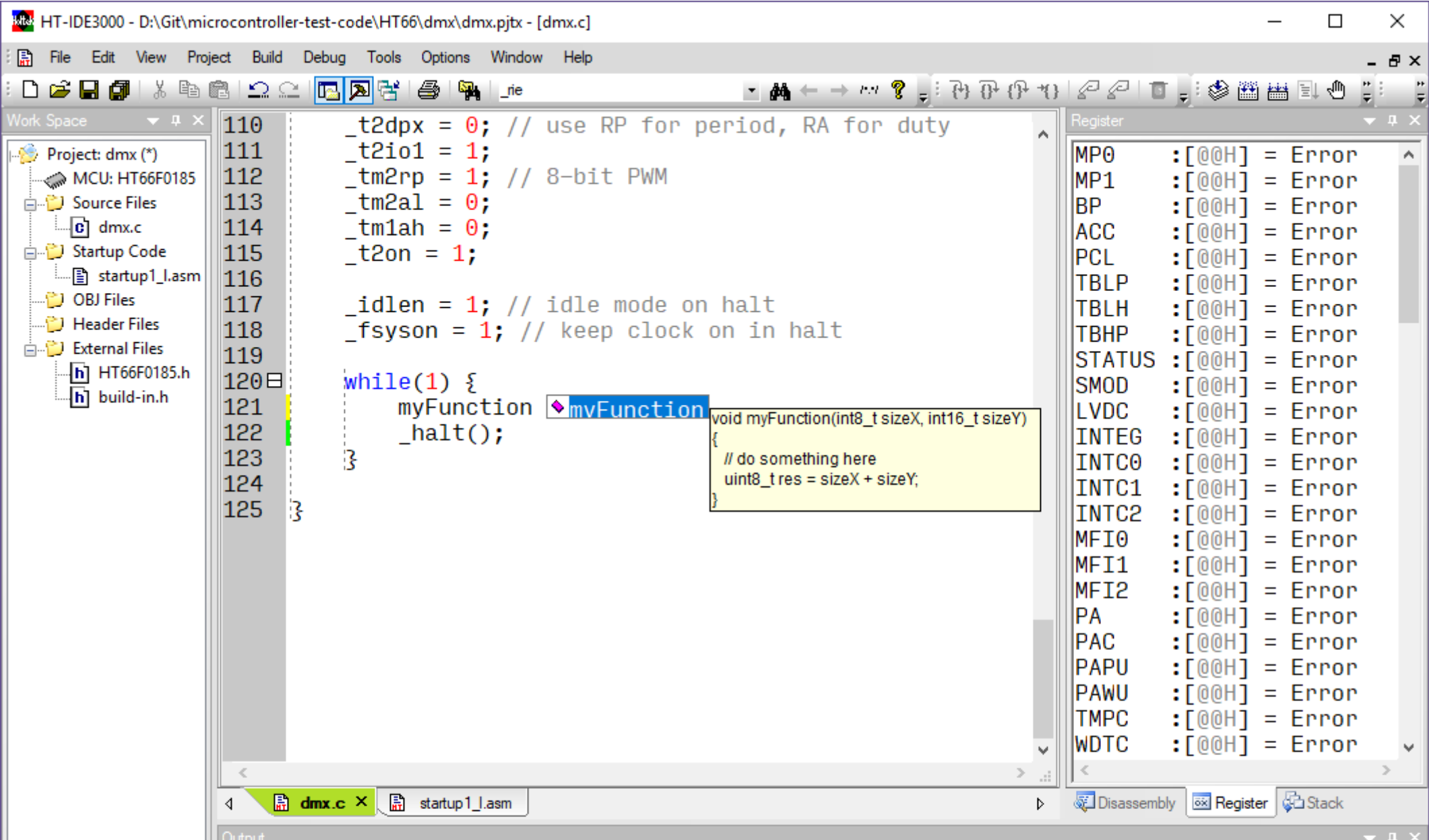
Basically, you're going to have to have your datasheet open so that you can hand-decode the values of these registers, as Microchip doesn't seem interested in integrating that style of documentation into their IDE.

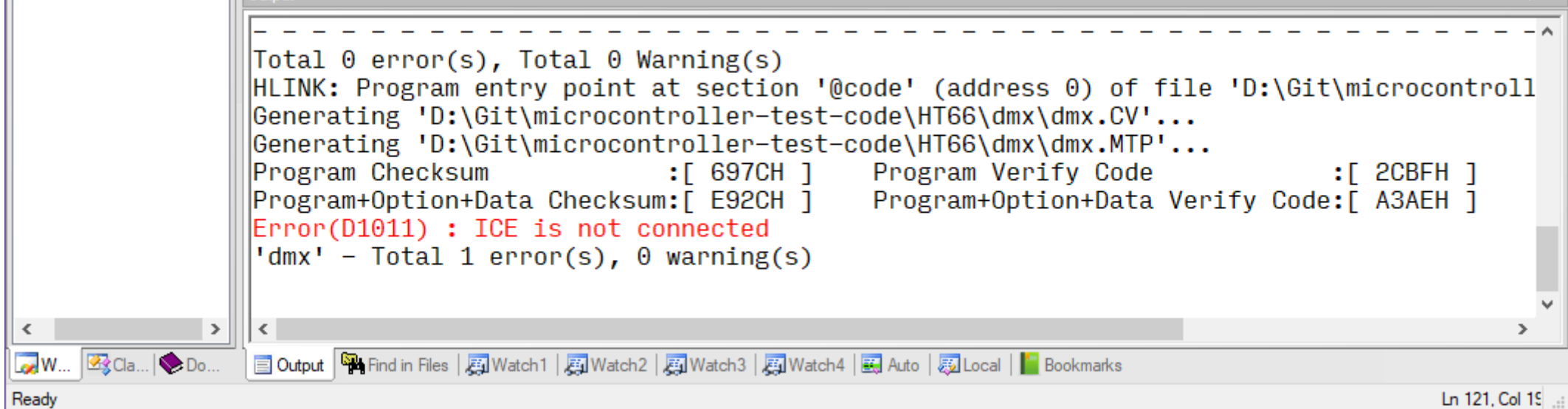
```
51
52 void __attribute__ ((vector(_UART1_ERR_VECTOR)))
53 {
54     [U1STAbits] - Not Recognized
55     if ((U1STA)its.OERR == 1)
56     {
57         U1STACLRL = _U1STA_OERR_MASK;
58     }
59 }
```

MPLAB X's NetBeans backend seems confused by bit-field register definitions, which makes hover-over symbol inspection basically worthless when you're in a debug session.

These are the things that drive me nuts – it's 2017; all of this data is already computerized. All they need to do is add a bit more functionality to their view (which many Eclipse-based vendor tools have), and all the sudden, the SFR view becomes ten times more productive to use.

For higher-end PIC32 and PIC24 devices, there's a semihosting-like feature Microchip calls appIO (though it only works with their pricy ICD debuggers). There's also runtime variable watch and instruction tracing, but that's only supported by the *really* pricy RealICE debugger.





```
Total 0 error(s), Total 0 Warning(s)
HLINK: Program entry point at section '@code' (address 0) of file 'D:\Git\microcontroll
Generating 'D:\Git\microcontroller-test-code\HT66\dmx\dmx.CV'...
Generating 'D:\Git\microcontroller-test-code\HT66\dmx\dmx.MTP'...
Program Checksum           :[ 697CH ]      Program Verify Code           :[ 2CBFH ]
Program+Option+Data Checksum:[ E92CH ]      Program+Option+Data Verify Code:[ A3AEH ]
Error(D1011) : ICE is not connected
'dmx' - Total 1 error(s), 0 warning(s)
```

The screenshot shows the HT-IDE3000 interface with a toolbar at the bottom containing icons for 'W...', 'Cla...', 'Do...', 'Output', 'Find in Files', 'Watch1', 'Watch2', 'Watch3', 'Watch4', 'Auto', 'Local', and 'Bookmarks'. The status bar at the bottom left shows 'Ready' and the bottom right shows 'Ln 121, Col 19'.

HT-IDE3000 looks like Office 2003's ugly cousin, but its ancient GUI toolkit snaps along on modern systems at lightning speed. Text completion is basic but rocket-fast, and the IDE integrates well with the debugger and target MCU.

HOLTEK HT-IDE3000

Anyone who complains about IDE bloat and download registration walls should immediately check out Holtek HT-IDE3000 (yes, that's what it's called). [Visit their website](#), click the download link, and a small 90 MB ZIP file starts downloading.

They do make you type in the serial number on your \$40 eLink debugger before using it (likely to deter debugger cloning), but that's the only registration you'll see.

Once you get the IDE up and running, the first thing you'll notice is its Office 2003 look. It ain't pretty, but it's functional. All the menu buttons you see are completely customizable through a drag-and-drop editor, which I didn't see in any other IDE tested.

The best part about having such an old-school IDE is that this thing *screams*: zero-lag text-editing, immediate hover-over pop-up tooltips with function source code (like the Hover Source feature in Eclipse, but much more lightweight), and zippy project building. I did a double-take when looking at RAM usage: *HT-IDE3000 uses 9.9 MB of RAM*. Yes, that decimal point is placed properly. Insane.

This comes at the expense of code intelligence features. IDE3000 can go to variables, macros, and function definitions; auto-complete variables and functions; and display pop-up source listings of functions when you hover over them. But there are no real macro expansion capabilities or code outline views, and the text completion is pretty basic: it recognizes C types and functions but doesn't seem to like remembering SFRs or other types of things that get `#define'd`. There's no macro expansion, and the text completion isn't intelligent – it's only a selectable list of all symbols visible in the given context.

This turns out to be good enough for the sorts of projects these small devices tackle. It's not like you're going to be juggling around more than a few dozen variables, and I'd bet 90% of the code-bases that target these parts comprise a single C file.

What you *do* get with Holtek is deep integration with the device you're targeting. There's built-in functionality for interacting with EEPROM on the device, programming using USB bootloaders present in some of their chips, as well as a built-in OTP workflow for generating the appropriate files Holtek needs to manufacture your parts (since most Holtek devices sold are OTP parts).

The screenshot displays the HT-IDE3000 interface with the following components:

- Work Space:** Shows a project named 'dmx (*)' for an MCU: HT66F018. Source files include 'dmx.c', 'startup1.asm', and various header files.
- Code Editor:** Contains C code for an interrupt service routine. The current line is `_pa6 = 1;` at address 0078. Other code includes variable declarations for `address`, `data`, and `currentIndex`, and logic for handling interrupts and data storage.
- Program View:** Shows assembly instructions corresponding to the C code. The current instruction is `SET PA6` at address 0078.
- Register View:** Lists various registers and their values. Notable registers include `STATUS` (07H), `PA` (02H), and `LVRC` (55H).
- Auto Watch Window:** A table showing the current state of variables:

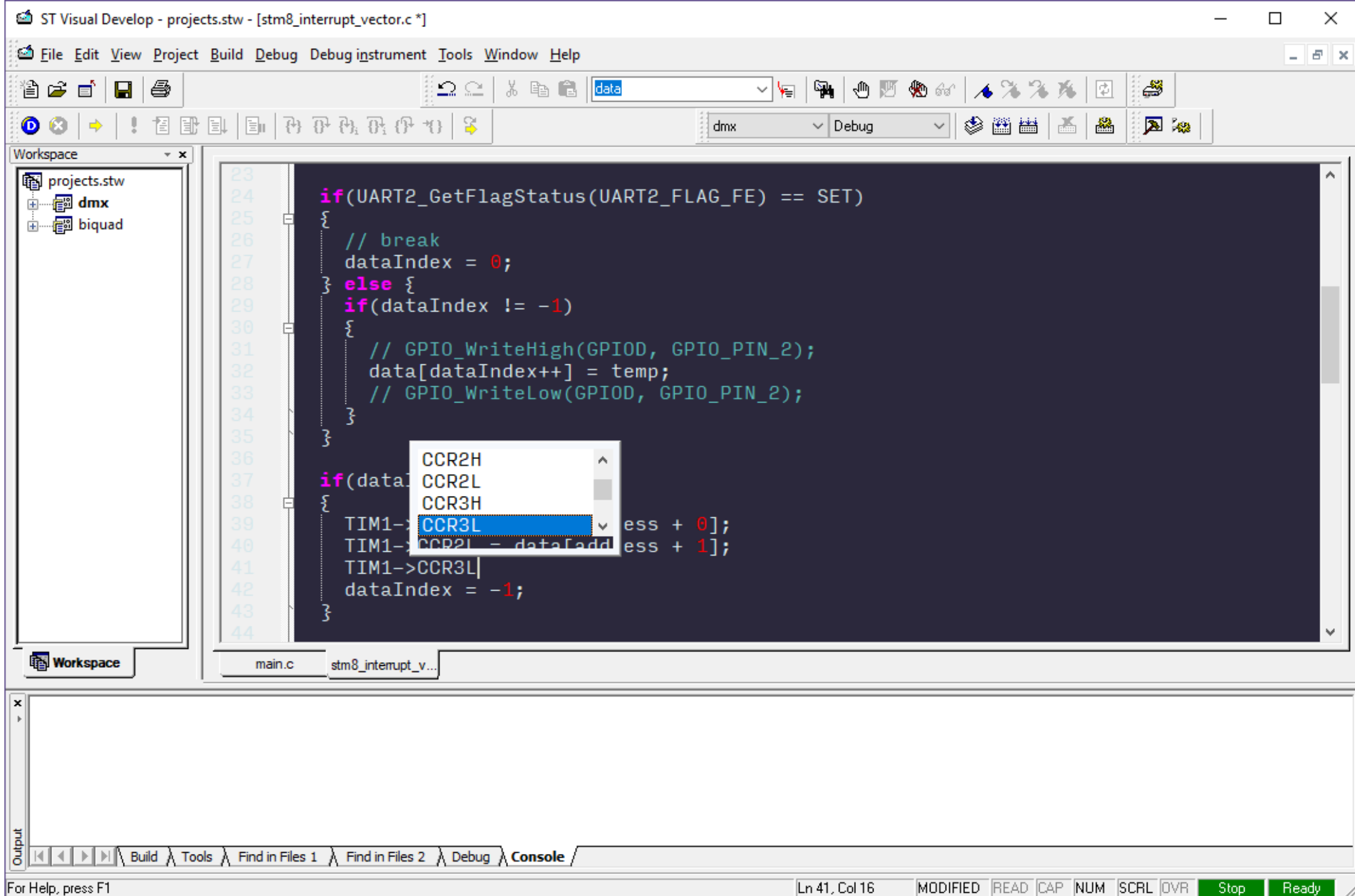
Name	Value	Address	Space
currentIndex	-1	0x0080	Ram
data	{...}	0x0083	Ram
data[0]	0 ''	0x0083	Ram
data[1]	0 ''	0x0084	Ram
data[2]	0 ''	0x0085	Ram
data[3]	0 ''	0x0086	Ram
data[4]	0 ''	0x0087	Ram
address	1 ''	0x0082	Ram

Debugging in HT-IDE3000 is lightweight and snappy, with most features you'd expect. I wish the register view had more information about the individual bits (at least their full name and description as a hover-over tooltip), but it gets the job done — especially considering how simple the architecture is this IDE targets.

As soon as you build a project, IDE3000 will jump into a debugging session in the background – even as you continue editing and building. This unconventional experience lures you tap that “Build” button often, and gush at your progress. As most firmware projects targeting this microcontroller will hover in a `while()` loop until interacted with, while you’re editing, you can double-click on a line of code to set a breakpoint, immediately check out what’s going on, make changes, and continue editing – without the manual process of uploading code, switching to a debug perspective, waiting for the image to flash, start running, and a breakpoint hit.

Code building and uploading are so fast, it almost feels like you’re in a simulator or a PC-based environment. The debugging views are basic but fully-functional. The peripheral register view will break down each register into the bits that are set and cleared – but I would have preferred hover-over descriptions of the registers and the bits they command.

While Holtek could modernize the UI, you’ll get no complaints about stability from me: I didn’t have a single crash, bug, hiccup, or driver installation kerfuffle the entire time I used IDE3000 – I tried it on both Windows 7 and Windows 10 Fall Creator’s Update, and the experience was identical. Holtek accelerates an update schedule by squashing bugs and adding new features several times a year – in fact, they had two updates over the course of my writing this review, which is more than any other IDE.



STVD supports decent text completion, and customizable editor colors and fonts.

ST VISUAL DEVELOP (STVD)

STVD is the official IDE for the [STM8](#). Its UI feels even older than IDE3000, but it's a bit better when it comes to text-editing capabilities. Its code completion is invoked with the familiar ctrl-spacebar shortcut, and instead of just displaying all symbols discovered, it seems to default to symbols that make sense in the present context. It knows enough about C to walk into pointer-referenced structs, but unfortunately, has no pop-up documentation or source-code browsing when you hover over a method.

Unfortunately, it feels slow. Code completion can take a second or longer to pop-up in larger projects, rearranging toolbars was sluggish, going to the definition of a symbol took time to load the editor window, and menus were slow to appear.

STM8S005K6 STM SWIM - projects.stw* - [Debug] - dmx.elf - [stm8_interrupt_vector.c]

File Edit View Project Build Debug Debug instrument Tools Window Help

Workspace

- projects.stw
 - dmx
 - FWLib
 - Source Files
 - main.c
 - stm8_interrupt_vector.c
 - Include Files
 - External Dependencies
 - biquad
 - FWLib
 - Source Files
 - Include Files
 - External Dependencies

```

17
18 @far @interrupt void UART2_RX_IRQHandler (
19     void)
20 {
21     uint8_t temp;
22     GPIO_WriteHigh(GPIOD, GPIO_PIN_0);
23     temp = UART2_ReceiveData8();
24
25     if(UART2_GetFlagStatus(UART2_FLAG_FE) ==
26     SET)
27     {
28         // break
29         dataIndex = 0;
30     } else {
31         if(dataIndex != -1)
32         {
33             // GPIO_WriteHigh(GPIOD, GPIO_PIN_2);
34             data[dataIndex++] = temp;
35             // GPIO_WriteLow(GPIOD, GPIO_PIN_2);
36         }
37     }
38
39     if(dataIndex > 400)
40     {
41         // ...
42     }
43 }

```

Disassembly

```

rupt_vector.c:21  GPIO_WriteHigh(GPIOD, G
0x871e <._RX_IRQHandler+19> 0x4B01
0x8720 <._RX_IRQHandler+21> 0xAE500F
0x8723 <._RX_IRQHandler+24> 0xCD823E
0x8726 <._RX_IRQHandler+27> 0x84
rupt_vector.c:22  temp = UART2_ReceiveDat
0x8727 <._RX_IRQHandler+28> 0xCD847D
0x872a <._RX_IRQHandler+31> 0x6B01
rupt_vector.c:24  if(UART2_GetFlagStatus(
0x872c <._RX_IRQHandler+33> 0xAE0002
0x872f <._RX_IRQHandler+36> 0xCD8481
0x8732 <._RX_IRQHandler+39> 0x4A
0x8733 <._RX_IRQHandler+40> 0x2603
rupt_vector.c:27  dataIndex = 0;
0x8735 <._RX_IRQHandler+42> 0x5F
0x8736 <._RX_IRQHandler+43> 0x200E
rupt_vector.c:29  if(dataIndex != -1)
0x8738 <._RX_IRQHandler+45> 0xCE0101
0x873b <._RX_IRQHandler+48> 0xA3FFFF
0x873e <._RX_IRQHandler+51> 0x2709
rupt_vector.c:32  data[dataIndex++] =
0x8740 <._RX_IRQHandler+53> 0x7B01
0x8742 <._RX_IRQHandler+55> 0xD70103
0x8745 <._RX_IRQHandler+58> 0x5C

```

Workspace

c:\Program Fil... stm8s_uart2.c stm8s_tim1.c main.c stm8_interrupt_...

Peripheral registers	Value
Port C	
Port D	
[0x500f] PD_ODR - Port D data output latch register	0x01
[0x5010] PD_IDR - Port D input pin value register	0xff
[0x5011] PD_DDR - Port D data direction register	0x01
[0x5012] PD_CR1 - Port D control register 1	0x03
[0x5013] PD_CR2 - Port D control register 2	0x00
Port E	
Port F	

Variable	Value	Type	Address
dataIndex	-1	short	0x101
data	\000' <repeats 512 times>	unsigned char [...]	0x103
[0]	0 '\000'	unsigned char	0x103
[1]	0 '\000'	unsigned char	0x104
[2]	0 '\000'	unsigned char	0x105
[3]	0 '\000'	unsigned char	0x106
[4]	0 '\000'	unsigned char	0x107
[5]	0 '\000'	unsigned char	0x108

Watch 1 Watch 2 Watch 3 Watch 4

For Help, press F1

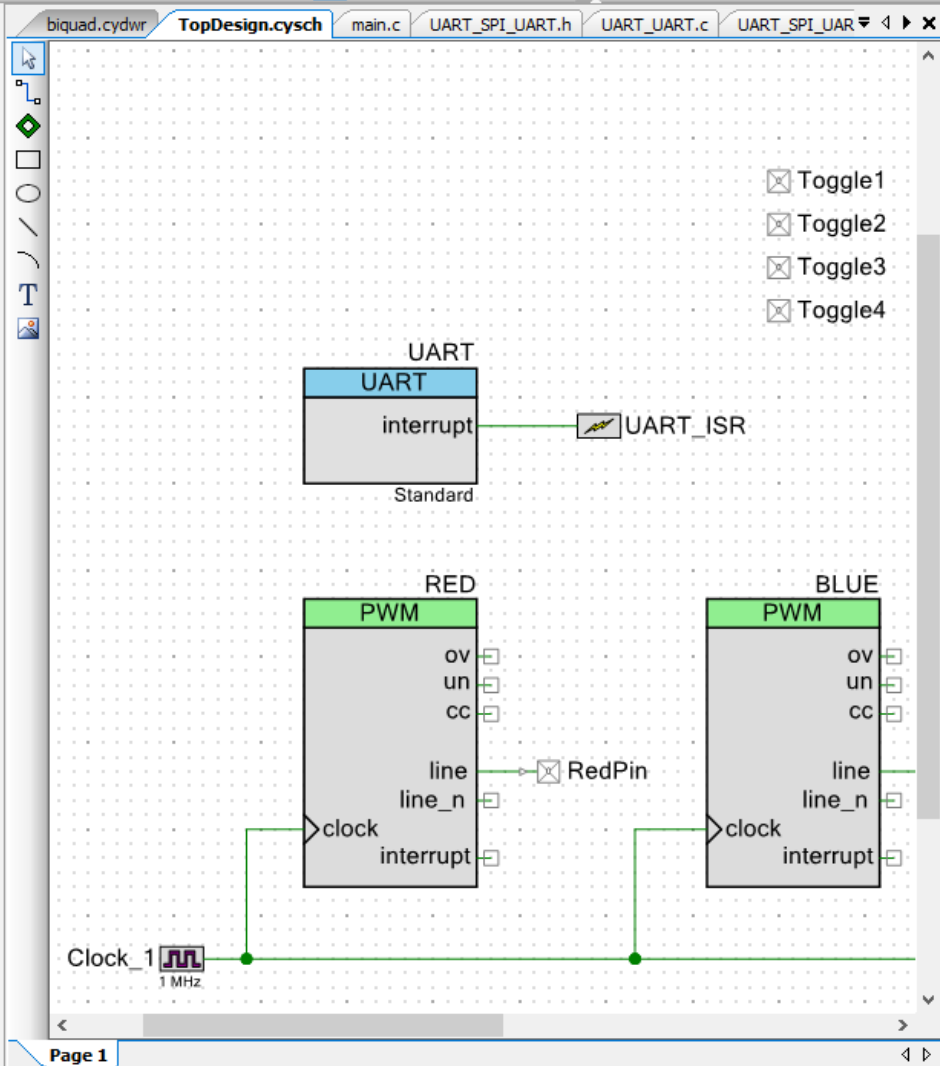
Ln. Col MODIFIED READ CAP NUM SCRL OVR Stop Ready

The peripheral registers view had descriptions of all the registers, but didn't break down registers into individual bits (or explain what they were).

Debugging was also worse than average. Breakpoints can't be set while the target is running – you must manually pause it, add your breakpoint, and then continue execution. The peripheral registers view was also underwhelming: while it has descriptions of registers, it only displays the whole value of the register – in hex format – without giving you a per-bit breakdown useful for catching bit math bugs in your code. Yet again, you'll be resigned to manually cross-checking register values with values from the datasheet – one of my biggest pet peeves.

Workspace Explorer (3 projects)

- *Workspace 'PSOC 4000S' (3 Projects)
 - Project 'biquad' [CY8C4045AZI-S413]
 - TopDesign.cysch
 - Project 'dmx' [CY8C4045AZI-S413]
 - Design Wide Resources (dmx.cydwr)
 - Pins
 - Analog
 - Clocks
 - Interrupts
 - System
 - Directives
 - Flash Security
 - Header Files
 - cyapicalbacks.h
 - Source Files
 - main.c
 - Generated_Source
 - PSoC4
 - BLUE
 - BluePin
 - Clock_1
 - cy_boot
 - cy_lfclk
 - GREEN
 - GreenPin
 - RED
 - RedPin
 - Toggle1
 - Toggle2
 - Toggle3
 - Toggle4
 - UART
 - UART.c
 - UART.h
 - UART_BOOT.c
 - UART_BOOT.h
 - UART_PINS.h
 - UART_PM.c
 - UART_PVT.h
 - UART_SPI_UART.c
 - UART_SPI_UART.h
 - UART_SPI_UART_INT.c
 - UART_SPI_UART_PVT.h
 - UART_UART.c
 - UART_UART_BOOT.c



Output

Show output from: All

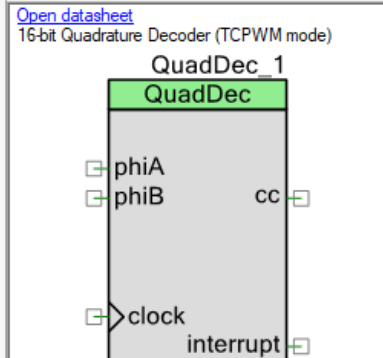
```
Erasing...
Programming of Flash Starting...
Protecting...
Verify Checksum...
Finished Programming
```

Component Catalog (89 components)

Search for...

Cypress Off-Chip

- Cypress Component Catalog
 - Analog
 - CapSense
 - Communications
 - Digital
 - Functions
 - PWM (TCPWM mode) [v2.10]
 - Quadrature Decoder (TCPWM mode) [v2.10]
 - Timer Counter (TCPWM mode) [v2.10]
 - Timer Counter PWM (TCPWM mode) [v2.10]
 - Logic
 - Digital Constant [v1.0]
 - Logic High '1' [v1.0]
 - Logic Low '0' [v1.0]
 - SmartIO [v1.10]
 - Virtual Mux [v1.0]
 - Display
 - Ports and Pins
 - Analog Pin [v2.20]
 - Digital Bidirectional Pin [v2.20]
 - Digital Input Pin [v2.20]
 - Digital Output Pin [v2.20]
 - System
 - Bootloadable [v1.50]
 - Bootloader [v1.50]
 - Clock [v2.20]
 - Global Signal Reference [v2.10]
 - Interrupt [v1.70]



System	
Interrupts	1 / 16
IO	8 / 36
Segment LCD	0 / 1
CapSense	0 / 1
Communication	
Serial Communication (SCB)	1 / 2
Digital	
Timer/Counter/PWM	3 / 5
Smart IO Ports	0 / 2
Analog	
Comparator	0 / 1
LP Comparator	0 / 2
DAC	0 / 2
Memory	
Flash	7.8 %
SRAM	47.9 %

PSoC Creator is built around Cypress's code-gen tools, which provides a schematic capture interface for instantiating and connecting components; these tools generate the configuration bitstream along with an API for the peripherals.

PSOC CREATOR

Cypress PSoC Creator is the official – and only – development environment available for Cypress's line of PSoC devices, and encompasses a project management system, text editor, schematic capture tool, code generator, and debugger.

PSoC Creator's interface lurks out of the last decade with its Office 2003 costume, and its .NET codebase ensures porting it to Linux or macOS will be an arduous task that Cypress, itself, resigns as unlikely.

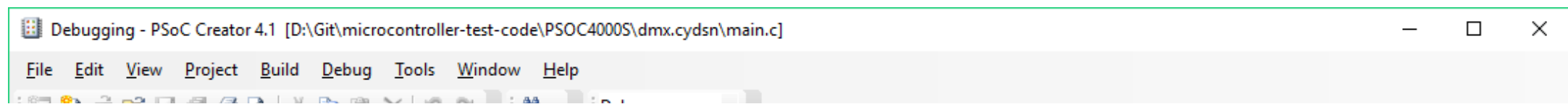
PSoC Creator feels snappier and lighter-weight than Eclipse or NetBeans-based environments while maintaining 90% of the IDE features and code introspection abilities. Loadable workspaces weave together one or more projects – any number of which can be open at the same time – exactly as Eclipse does.

A Code Explorer pane leads you to variable and function definitions with a single click; while the text competition digs deep into functions, macros, and variables to bubble up suggestions with the familiar Ctrl-Spacebar shortcut.

Cypress needs to refashion hover-over tooltips into what Eclipse or IDE3000 does; as it stands, the tooltips print the name of the function and its parameters – there are no code docs, and no quick source view you can use to peek at a function's implementation.

There's one-click documentation access, plus a resource meter for monitoring both flash and SRAM usage – but also peripheral usage, which shows you exactly how integrated PSoC Creator is with the underlying PSoC hardware.

The 500 lb gorilla in the PSoC Creator ecosystem – the schematic capture configurator tool – is a major omnipresence in PSoC Creator – but I'll save that discussion for Code Generator section of this review page.



RED Debug Window

COUNTER = 0x00000084
 CC = 0x00000023
 CC_BUF = 0x0000FFFF
 PERIOD = 0x000000FF
 PERIOD_BUF= 0x0000FFFF

GREEN Debug Window

COUNTER = 0x000000C5
 CC = 0x00000080
 CC_BUF = 0x0000FFFF
 PERIOD = 0x000000FF
 PERIOD_BUF= 0x0000FFFF

BLUE Debug Window

COUNTER = 0x0000009E
 CC = 0x00000080
 CC_BUF = 0x0000FFFF
 PERIOD = 0x000000FF
 PERIOD_BUF= 0x0000FFFF

UART Debug Window

_CTRL =
 _SPI_CTRL =
 _SPI_STATUS =
 _UART_CTRL =
 _UART_TX_CTRL =
 _UART_RX_CTRL =
 _UART_FLOW_CTRL =
 _I2C_CTRL =
 _I2C_STATUS =
 _I2C_M_CMD =

```

36     uint8_t temp = UART_SpiUartReadRxData();
37     if(idx >= 0)
38         data[idx++] = temp;
39
40     if(idx > 512)
41     {
42         //
43         Toggle4_Write(1);
44         RED_WriteCompare (data[address+0]);
45         GREEN_WriteCompare (data[address+1]);
46         BLUE_WriteCompare (data[address+2]);
47         idx = -1;
48     }
49     //
50     Toggle3_Write(0);
51 }
52
53 UART_ClearRxInterruptSource (source);
54 source = UART_GetRxInterruptSource ();
55 Toggle1_Write(0);
56 //Toggle3_Write(0);
57 }
58
59 int main(void)
60 {
61     RED_Start();
62     GREEN_Start();
63     BLUE_Start();
64
65     UART_ISR_StartEx (UartInterrupt);
66
67     UART_Start();
  
```

Code Explorer (main.c)

Include directives:
 project.h

Global Variables:
 address : uint8_t
 idx : int16_t
 data : uint8_t [513]

Function definitions:
 UartInterrupt() : void
 source : uint32_t
 temp : uint8_t
 main() : int

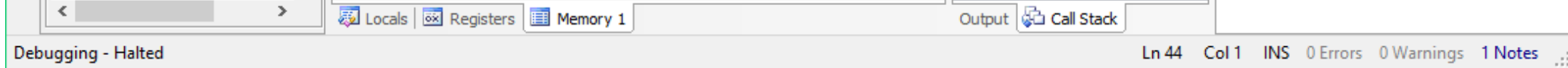
Memory 1

Address: 0x00000000 Address Space: All

0x00000000	00	10	00	20	11	00	00	...
0x00000008	31	02	00	00	31	02	00	1...1...
0x00000010	10	b5	00	f0	fb	fa	00	f0
0x00000018	13	f9	c0	46	10	b5	72	b6
0x00000020	09	22	1f	4b	1f	4c	1a	60
0x00000028	1f	4a	20	4b	20	20	1a	60

Call Stack

Level	Function
0	UartInterrupt()
1	<signal handler called>()
2	CyExitCriticalSection()
3	CySysPmSleep()
4	main()



PSoC Creator has no global register inspector like other environments; you can only view peripheral registers associated with instantiated components.

Flash load times were unimpressive (though not the worst seen in my round-up). A basic, 1.4 KB program took 7.82 seconds to load and run to main(). Filling flash up to 16 KB took 10.84 seconds.

PSoC Creator jostled my brain when I strained to reduce power consumption in my DMX project. I couldn't find a standard list of all peripheral registers which would allow me to cross-reference bits from the datasheet – instead, you can select from the peripherals you've configured in the schematic capture view, which initiates a limited view of that specific peripheral (and only displaying the registers associated with its current function).

As this is an ARM microcontroller, you may intuit that a cross-platform GCC-based toolchain would be easy to set up – but because of the proprietary configuration bitstream required when flashing these devices, you'll need to use Cypress's tools to fit your design to the processor and merge this bitstream with your application binary code.

While Cypress stops short of supporting Linux or macOS outright, PSoC Creator can export projects to common cross-platform Eclipse or Makefile-based environments – as well to industry stalwarts μ Vision and IAR.

But if you're on Windows, I wouldn't bother – this is as close to an Eclipse/NetBeans workflow you'll get with a proprietary IDE, and it's definitely good enough to compete with either.

KEIL μ VISION

Keil's fame ensues from their acclaimed 8051 compiler, C51, introduced in 1988 – but since 1997, their compilers have also shipped with their in-house IDE, Keil μ Vision.

Now at version 5, μ Vision continues to be popular among professional dev shops for both Arm and 8051 development, even as more and more manufacturers roll their own cross-platform toolchains.

Speaking of that, while every Arm part I reviewed has its own manufacturer-provided IDE (save for the M051), developers can use μ Vision as an officially-supported development environment for all them.

On the other end of the spectrum, among the 8051, only Silicon Labs maintains their own IDE for the EFM8 – both the N76 and STC8 only support μ Vision. Keil's IDE has support for multi-project workspaces, though it doesn't enforce their use.

Keil's text editor is painful to use. The first two things you'll notice are that there are no auto-indenting inside block statements, and out of the box, there's no keyboard shortcut for commenting or uncommenting code. Sure, you can bind "comment text selection" and "uncomment text selection" to shortcut keys, but you have to select the text before invoking. This is ridiculous.

Even more painful: μ Vision has a Jekyll-and-Hyde act when switching between C51 (8051) and MDK (Arm) projects. When building Arm projects, μ Vision has ok-but-not-great text-completion (in the form of a pop-up list of discovered symbols) – but when building 8051 projects, that text completion inexplicably vanishes.

D:\Git\microcontroller-test-code\M0516LAN\projects\toggle\toggle.uvprojx - µVision

File Edit View Project Flash Debug Peripherals Tools SVCS Window Help

Registers Disassembly GP

Register	Value
R0	0x00000008
R1	0x00000001
R2	0x00000001
R3	0x50004000
R4	0x000006E4
R5	0x20000008
R6	0x00000000
R7	0x00000000
R8	0xFFFFFFFF
R9	0xFFFFFFFF
R10	0x000006E4
R11	0x000006E4
R12	0xFFFFFFFF
R13 (SP)	0x20000268
R14 (LR)	0x000005...
R15 (PC)	0x000005...
xPSR	0x61000000
Banked	
System	
Internal	
Mode	Thread
Stack	MSP

```

0x000005D0 4806 LDR r0,[pc,#24] ; @0x000005EC
0x000005D2 F7FFF8A BL.W __semlibhosting_library_function (0x00
9: while(1) {
0x000005D6 E008 B 0x000005EA
10: P00 = !P00;
0x000005D8 4805 LDR r0,[pc,#20] ; @0x000005F0
0x000005DA 6800 LDR r0,[r0,#0x00]
0x000005DC 2800 CMP r0,#0x00
0x000005DE D101 BNE 0x000005E4
  
```

```

toggle.c SysInit.c gpio.h M051Series.h gpio.c
1 #include "M051Series.h"
2
3 extern void SYS_Init(void);
4
5 int main()
6 {
7     SYS_Init();
8     GPIO_SetMode(P0, BIT0, GPIO_PMD_OUTPUT);
9     while(1) {
10        P00 = !P00;
11    }
12 }
  
```

Property	Value
P0_PMD	0x0000FFFF
PMD0	1: 1 = Px [n] pin is in OUTPUT mode
PMD1	0: 0 = Px [n] pin is in INPUT mode
PMD2	1: 1 = Px [n] pin is in OUTPUT mode
PMD3	2: 2 = Px [n] pin is in Open-Drain mode
PMD4	3: 3 = Px [n] pin is in Quasi-bidirectional mode
PMD5	3: 3 = Px [n] pin is in Quasi-bidirectional mode
PMD6	3: 3 = Px [n] pin is in Quasi-bidirectional mode
PMD7	3: 3 = Px [n] pin is in Quasi-bidirectional mode
P0_OFFD	0
P0_DOUT	0x000000FF
P0_DMASK	0
P0_PIN	0x000000FF
P0_DBEN	0
P0_IMD	0
P0_JEN	0
P0_JSRC	0
P0_PMD	0x0000FFFF

PMD0
[Bits 1..0] RW (@ 0x50004000)
Px I/O Pin[n] Mode Control
Determine each I/O type of Px pins

Command

```

*** Restricted Version with 32768 Byte Code Size Limit
*** Currently used: 1772 Bytes (5%)

BS \\toggle\toggle.c\7
  
```

Call Stack + Locals

Name	Location/Value	Type
main	0x00000000	int f()

ASSIGN BreakDisable BreakEnable BreakKill BreakList BreakSet BreakAccess

Keil µVision's Arm debugging has a functional peripheral inspector that breaks each register into each logical field, with named descriptions of any enumeration values.

Debugging is the same sad song. Keil has good support for semihosting in Arm projects without needing much user intervention. A decent peripheral register explorer helps you diagnose peripheral issues with named-value fields that have nice descriptions.

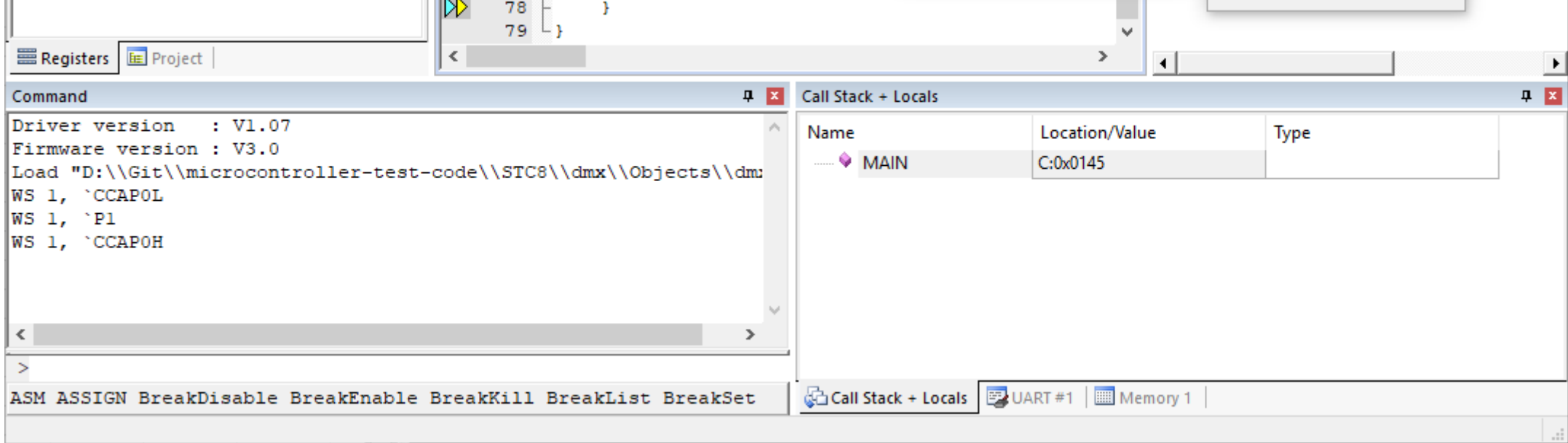
The screenshot shows the Keil uVision IDE interface. The main window displays a disassembly view of code, with the following assembly instructions visible:

```
68: TR1 = 1; //Timer1 running
69:
70: // UART2_INT_ENABLE(); // enable serial port
71: // S2_Int_Enable();
185: D28E SETB TR1(0x88.6)
186: S1_Int_Enable();
187: D2AC SETB ES(0xA8.4)
188: EA = 1; // enable interrupts
189: D2AF SETB EA(0xA8.7)
190: PCON |= SCON_SMOD0; // enable framing error
191: 438740 ORL PCON(0x87), #0x40
192: while(1)
193: 80FE SJMP C:018E
194: FF MOV R7,A
195: FF MOV R7,A
196: FF MOV R7,A
197: FF MOV R7,A
```

Overlaid on the IDE are several dialog boxes for configuring peripherals:

- Other**: A dialog box for general peripheral settings with fields for AUXR (0x40), PCON (0x70), AUXR1 (0x40), CLK_DIV (0x00), AUXR2 (0x00), and BUS_SPEED (0x00).
- Serial Channel**: A dialog box for serial port configuration. Mode is set to "8-Bit var.Baudrate". SCON is 0x50, SADDR is 0x00, SBUF is 0x00, and SADEN is 0x00. Checkboxes for SM2, TB8, RB8, SMOD0, FE, and REN are present. Baudrate settings for SMOD1, RCLK, and TCLK are also shown.
- CCP/PCA/PWM**: A dialog box for configuring the CCP module. Fields include CCON (0x40), CCAPM0 (0x42), CCAPM1 (0x42), CCAPM2 (0x42), CMOD (0x08), PCA_PWM0 (0x00), PCA_PWM1 (0x00), PCA_PWM2 (0x00), CH (0xDE), CCAP0H (0x00), CCAP1H (0x00), CCAP2H (0x00), CL (0x52), CCAP0L (0x00), CCAL1L (0x00), and CCAP2L (0x00).
- Parallel Port 0**: A dialog box for configuring Port 0. P0 is set to 0xFF, and all 8 bits are checked.
- Parallel Port 1**: A dialog box for configuring Port 1. P1 is set to 0xFF, and all 8 bits are checked.
- UART**: A dialog box for configuring the UART module. Fields include SADEN (0x00), SADDR (0x00), SCON (0x50), SBUF (0x00), S2CON (0x40), S2BUF (0x00), S3CON (0x40), S3BUF (0x00), S4CON (0x40), and S4BUF (0x00).

The background shows the Registers window on the left and the Symbols window on the right, which lists the DMX module.



Debugging on the 8051 is a different story — the pop-up peripheral inspectors are clunky, and not every vendor adds the proper support to Keil.

But in 8051land, weird pop-up peripheral inspectors stand in for the peripheral pane. While these curtail my productivity, I must admit they are more amusing than their Arm brethren (let's be honest: who doesn't love clicking checkboxes and watching LEDs light up?).

I scoured the menus, feature lists, help files, and online examples trying to redeem μ Vision by finding something it excelled at compared to the other IDEs I evaluated. I suppose it's much lighter-weight than Eclipse is — coming in at 105 MB of RAM usage during a debug session. It makes too many concessions for me to even consider it alongside Eclipse, though.

In the end, consider μ Vision as nothing more than the free editor you get when you download C51 or MDK-ARM. And it's not worth a penny more.

CODE GENERATION TOOLS

Peripheral configuration and bring-up is generally a drop in the bucket when compared to the time required to implement an entire commercial embedded project – but if you're working on tiny projects (either hobbyist stuff, or simple proof-of-concept engineering demos), having a code-gen tool can noticeably speed up the development cycle.

Many of the development environments tested have code-gen tools either integrated directly into the IDE (Microchip Code Configurator, Simplicity Configurator, Infineon DAVE, PSoC Creator, Processor Expert, or have stand-alone tools (STM32CubeMX, Atmel START, STC-ISP, MCUXpresso Config Tools).

Some generate initialization code that calls into the vendor's general-purpose peripheral libraries (like STM32CubeMX), while some generate raw register initialization values from scratch (Silicon Labs' Simplicity Configurator).

[VIEW CODE](#)[SAVE CONFIGURATION](#)[EXPORT PROJECT](#)

CLOCK CONFIGURATOR

[+ Zoom in](#) [- Zoom out](#) [Reset](#) [Show All Paths](#)

OSCILLATORS

- External Crystal Oscillator 0.4-32MHz (XOSC)
- 32kHz External Crystal Oscillator (XOSC32K)
- 32kHz High Accuracy Internal Oscillator (OSC32K)
Frequency: 32.768 kHz
- 8MHz Internal Oscillator (OSC8M)
- Digital Frequency Locked Loop (DFLL48M)
Frequency: 48 MHz
- Fractional Digital Phase Locked Loop (FDPLL96M)
- 32kHz Ultra Low Power Internal Oscillator (OSCULP32K)
Frequency: 32.768 kHz

SOURCES

- Generic clock generator 0
Frequency: 48 MHz
- Generic clock generator 1
Frequency: 32.768 kHz
- Generic clock generator 2
- Generic clock generator 3
- Generic clock generator 4
- Generic clock generator 5

[Reset clock settings](#)CPU
Frequency: 48 MHzCOMPONENTS [+](#)

While Atmel START provides a nice-looking graphical interface especially useful for the flexible clocking schemes available on modern ARM microcontrollers, its design validation is hit or miss: in this picture, there's absolutely no warning that the main 32 kHz oscillator isn't enabled. Most

seriously, the DFLL module indicates it's outputting a frequency of 48 MHz, even though its multiplier is set to "0" in the configuration properties dialogue. Changing this to arbitrary values does not update the "48 MHz" display. This is a tool you cannot trust.

ATMEL START

Microchip's code-gen tool for its Atmel acquisition is a hold-over called Atmel START; it's a web-based code configurator that supports nearly their whole catalog of current devices. Under the hood, it generates peripheral libraries that are considered part of ASF4 (Atmel... err, "Advanced" ... Software Framework). This is a marked improvement over ASF3 on paper, but there are still significant problems with it.

First, the good: using the part with 8-bit megaAVR and tinyAVR devices is a no-brainer. You can use the "initialization" drivers instead of the "basic" drivers to generate init-only code, along with any stubbed out ISR functions you need. The generated code is beautifully documented, extremely readable, and compact. It's written the way you would probably write it by hand, and I can imagine this would serve as a nice learning tool for someone coming from Arduino, looking to take baby steps toward programming MCUs at the register level.

Now the bad: Atmel START is extremely clunky to use. Projects must originate in the web browser, then get exported to a ZIP archive format, where Atmel Studio extracts them to a solution, where you can then open them. If you want to change something, thankfully Atmel Studio allows you to right-click on the project and reconfigure it – opening a browser window inside the app and automatically loading the project.

But this begs the question: why can't Atmel Studio create these projects from the get-go, if it can edit and re-generate them? Another problem with Atmel START is the lack of placeholder markers the tool can use to determine what to preserve and what to overwrite – this is only done at a whole file level. I assume this is why they don't pre-define user callback function stubs, as these would get overwritten on subsequent regenerations of the code. This is a lazy design decision that makes integrating code with Atmel START challenging.

More bad news for START: Atmel's SAM D-series parts I evaluated require lots of clock configuration (like most Arm parts), but START provides absolutely no error-checking or clocking suggestions for you. It will gladly generate code that uses the main DFLL, without actually enabling the main DFLL, causing your program to hang in a while() loop waiting for the DFLL to start up, when it never will.

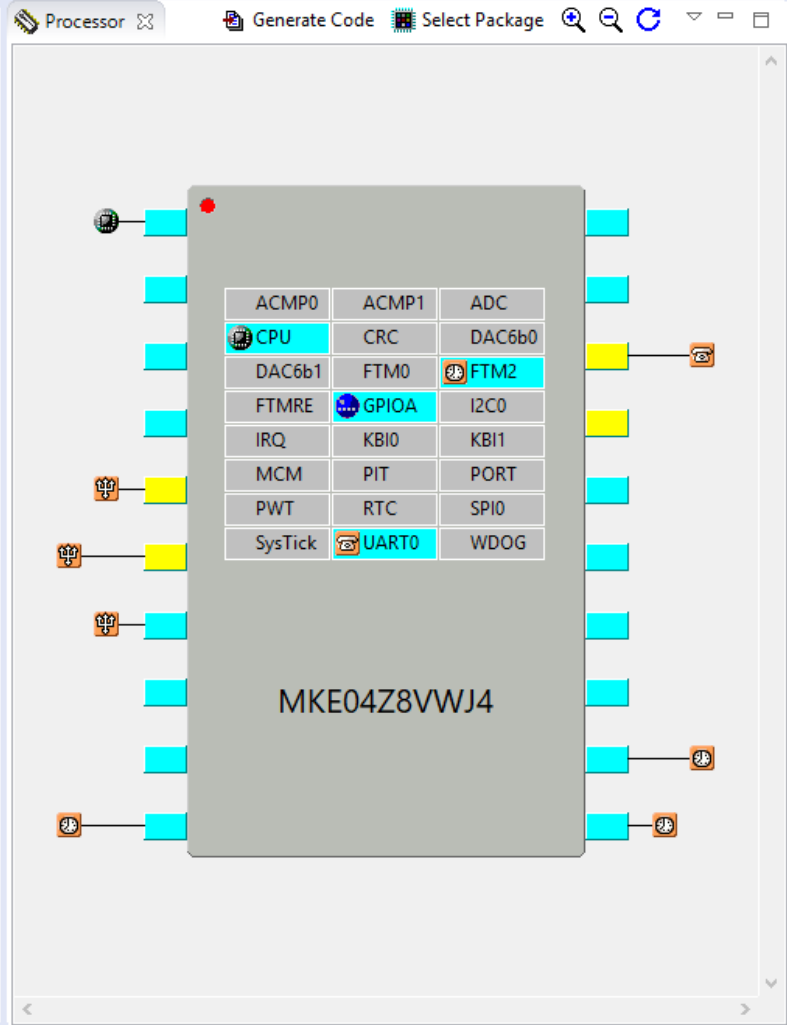
This is the crap that makes getting going on a new platform so challenging, and this is precisely why I like using code configuration tools. So if a tool can't do that, I scarcely see much value to it that I can't get from normal peripheral libraries.



Quick Access

C/C++ Hardware Debug

- Generator_Configurations
- FLASH
- OSs
- Processors
 - Cpu:MKE04Z8VWJ4
- Components
 - Referenced_Componer
 - UART:AsynchroSerial
 - RED:PWM
 - GREEN:PWM
 - BLUE:PWM
 - BreakReceived:BitIO
 - ByteReceived:BitIO
 - BufferFull:BitIO
- PDD



Component Inspector - UART

- Properties Methods Events
- type filter text
- ▼ All
 - ▼ Interrupt service/event
 - Handshake
 - ▼ Settings
 - Receiver
 - Transmitter
 - Initialization
 - CPU clock/speed selection
 - Referenced components

Components Library Basic Advanced

Component name:

Channel:

Settings Initialization CPU clock/speed sele >>2

Parity: none

Width: 8 bits

Stop bit: 1

Receiver

RxD:

RxD pin signal:

Transmitter

TxD:

TxD pin signal:

Baud rate: 250000 baud

Break signal:

Wakeup condition:

Transmitter output:

Receiver input:

Stop in wait mode:

Idle line mode:

Break generation length:

Problems Console

CDT Build Console [dmx]

```
Building file: ../Generated_Code/Vectors.c
Invoking: Cross ARM C Compiler
arm-none-eabi-gcc -mcpu=cortex-m0plus -mthumb -O3 -fmessage-length=0 -fsigned-char -ffunction-sections -fdata-sections -flto -g3 -I"D:/Git/microcontroller-test-code/KE04/dmx/Static_Code/PD
Finished building: ../Generated_Code/Vectors.c
```

Building target: dmx.elf



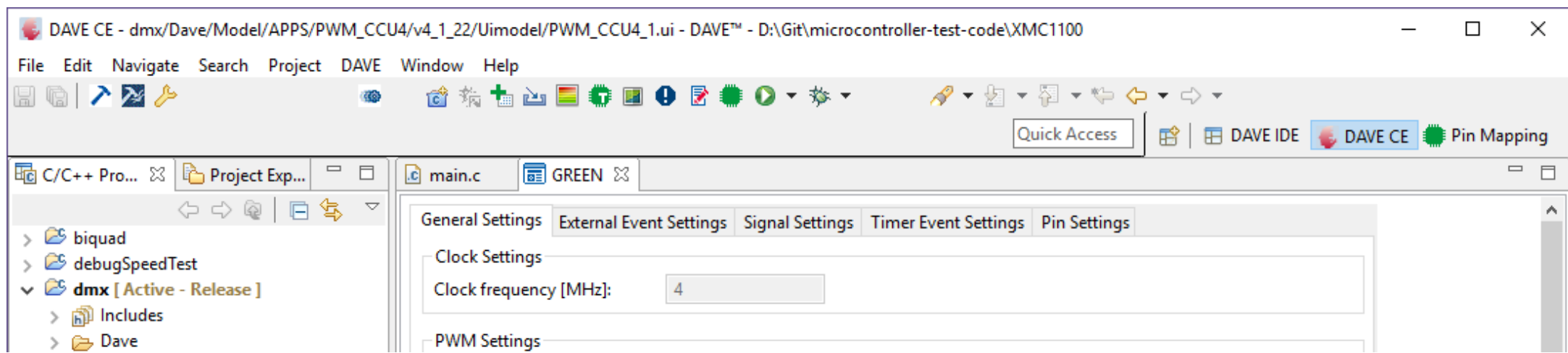
Processor Expert provides a stunning array of configuration options and flexibility — but make sure you have a fast desktop PC, as this tool is a huge resource hog. It also generated some of the slowest peripheral code in the round-up.

PROCESSOR EXPERT

Processor Expert is at the other end of the spectrum. PE generates initialization code, interrupts, user callbacks, linker files, and an entire peripheral library. It uses a component-oriented model with dependency resolution; for example, two high-level “PWM” component instances will share a common “Timer Unit” low-level component (as they will end up on different output-compare channels of that timer unit).

High-level components implement conceptual functionality, not peripheral functions. For example, if you wanted a function to execute every 25 ms, you would add a “TimerInt” component, and set the interval to 25 ms. Processor Expert will figure out which timer to use (FlexTimer, LPTimer, PIT, etc), route the clock appropriately, and calculate the necessary period register values, enable interrupts, generate an interrupt handler that takes care of any bits you need to set or clear. Of course, you can override any of its decisions at any point through its plethora of property panes — and unlike START, it will check your work to make sure everything is kosher.

While it’s extremely flexible, the biggest problem with Processor Expert is its interface is unbelievably slow — whenever you change a parameter, it can take several seconds (even on a fast computer) for that change to propagate. When you hit “Generate Code” prepare to wait 10-30 seconds at least — and project rebuilding takes forever, too.



The screenshot displays the Infineon DAVE IDE interface, which is used for configuring hardware and generating code. It is divided into several main sections:

- Project Explorer (Top Left):** Shows a file tree with folders for Libraries, Startup, and main.c. Files include startup_XMC1100.S, system_XMC1100.c, linker_script.ld, solver.bak, and toggle.
- Timer Settings (Top Right):** A configuration panel for a timer.

Counting mode:	Edge Aligned	<input checked="" type="checkbox"/> Start during initialization
		<input type="checkbox"/> Single-shot mode
Timer Settings		
PWM resolution [nsec]:	256	Actual PWM resolution [nsec]: 250
Frequency [Hz]:	100000	Actual frequency [Hz]: 100000
Duty cycle [%]:	100	Actual duty cycle [%]: 100
Prescaler:	0	Period register: 0x27
		Compare register: 0x0
- APP Dependency Tree (Middle Left):** A hierarchical tree view showing components like BLUE, DMX512_RD_0, GREEN, and RED, each with sub-components like GLOBAL_CCU4_0 and CLOCK_XMC1_0.
- HW Signal Connectivity (Bottom Right):** A diagram showing the interconnection of hardware components.
 - Inputs: DIGITAL_IO TOGGLE, PWM_CCU4 BLUE, PWM_CCU4 GREEN, PWM_CCU4 RED, DMX512_RD DMX512_RD_0, DIGITAL_IO DMX_INPUT.
 - Central Component: GLOBAL_CCU4 GLOBAL_CCU4_0.
 - Outputs: CLOCK_XMC1 CLOCK_XMC1_0, CPU_CTRL_XMC1 CPU_CTRL_XMC1_0.

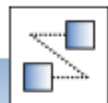
DAVE provides a property-based GUI code generator that integrates well into the IDE. The tool calls into a lightweight peripheral library, XMClib.

INFINEON DAVE

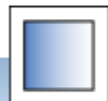
Infineon DAVE is similar from the user's perspective, but instead of generating nearly-unreadable, multi-level library code, DAVE generates initialization code calls into XMClib to manage peripherals. And unlike Processor Expert, DAVE designers appear to have designed the generated code (as well as XMClib in general) for better optimization than Processor Expert, where even a single bit-toggle function ends up nearly completely unoptimized – taking 40 cycles to complete.

DAVE also integrates higher-level “apps” – like built-in lighting control systems (including a DMX-512 receiver), communication protocols, graphics libraries, and motor controller libraries (we're talking about pretty advanced stuff: FOC of ACIMs and PMSMs).

DefaultMode Peripherals



Communications

 SMBus 0
 SPI 0
 UART 0


Core

 External Interrupts
 Flash Control
 Interrupts
 Reset Sources


Other

 CRC


Power

 PMU
 Supply Monitor
 Voltage Regulators


Timers

 PCA
 RTC
 Timers

- Mode Transitions
 - RESET → DefaultMode
- DefaultMode
 - Peripherals
 - Port I/O
 - CROSSBAR0
 - PB0
 - PB1
 - PB2

*Properti Periphera Search

Properties of Timers

TIMER Setup	TIMER 0/1	TIMER 2	TIMER 3
Property	Value		
<ul style="list-style-type: none"> <ul style="list-style-type: none"> Clock Control 0 <ul style="list-style-type: none"> Timer 0/1 Prescale 	SYSCLK / 4		
<ul style="list-style-type: none"> <ul style="list-style-type: none"> Timer 0 <ul style="list-style-type: none"> Mode Clock Frequency Clock Source Timer or Counter Timer Running State Timer Switch 1: Run Control Timer Switch 2: Gate Control 	Mode 2, 8-bit Cou 5.000 MHz Use the SCA presc Timer mode Timer is Running Start Disabled		
<ul style="list-style-type: none"> <ul style="list-style-type: none"> Timer 0 Firmware Control <ul style="list-style-type: none"> Timer Start or Stop Timer Interrupt Cleared by Timer Interrupt Enable Flag Timer Interrupt Pending Flag Timer Overflow Flag 	TR0(TCON.4) By Hardware ET0(IE.1) TF0(TCON.5) TF0(TCON.5)		
<ul style="list-style-type: none"> <ul style="list-style-type: none"> Timer 1 <ul style="list-style-type: none"> Mode Clock Frequency Clock Source Timer or Counter Timer Running State Timer Switch 1: Run Control Timer Switch 2: Gate Control 	Mode 2, 8-bit Cou 5.000 MHz Use the SCA presc Timer mode Timer is Running Start Disabled		
<ul style="list-style-type: none"> <ul style="list-style-type: none"> External Interrupt Settings <ul style="list-style-type: none"> INT0 Trigger Sensitivity INT1 Trigger Sensitivity 	Level-triggered Level-triggered		

Silicon Labs' Simplicity Studio includes the built-in Simplicity Configurator for generating lightweight initialization code and ISR stubs.

SILICON LABS SIMPLICITY CONFIGURATOR

Silicon Labs takes a much lighter-weight approach to code generation with Simplicity Configurator. All Simplicity Configurator does is create peripheral initialization code, by generating functions that directly initialize register values.

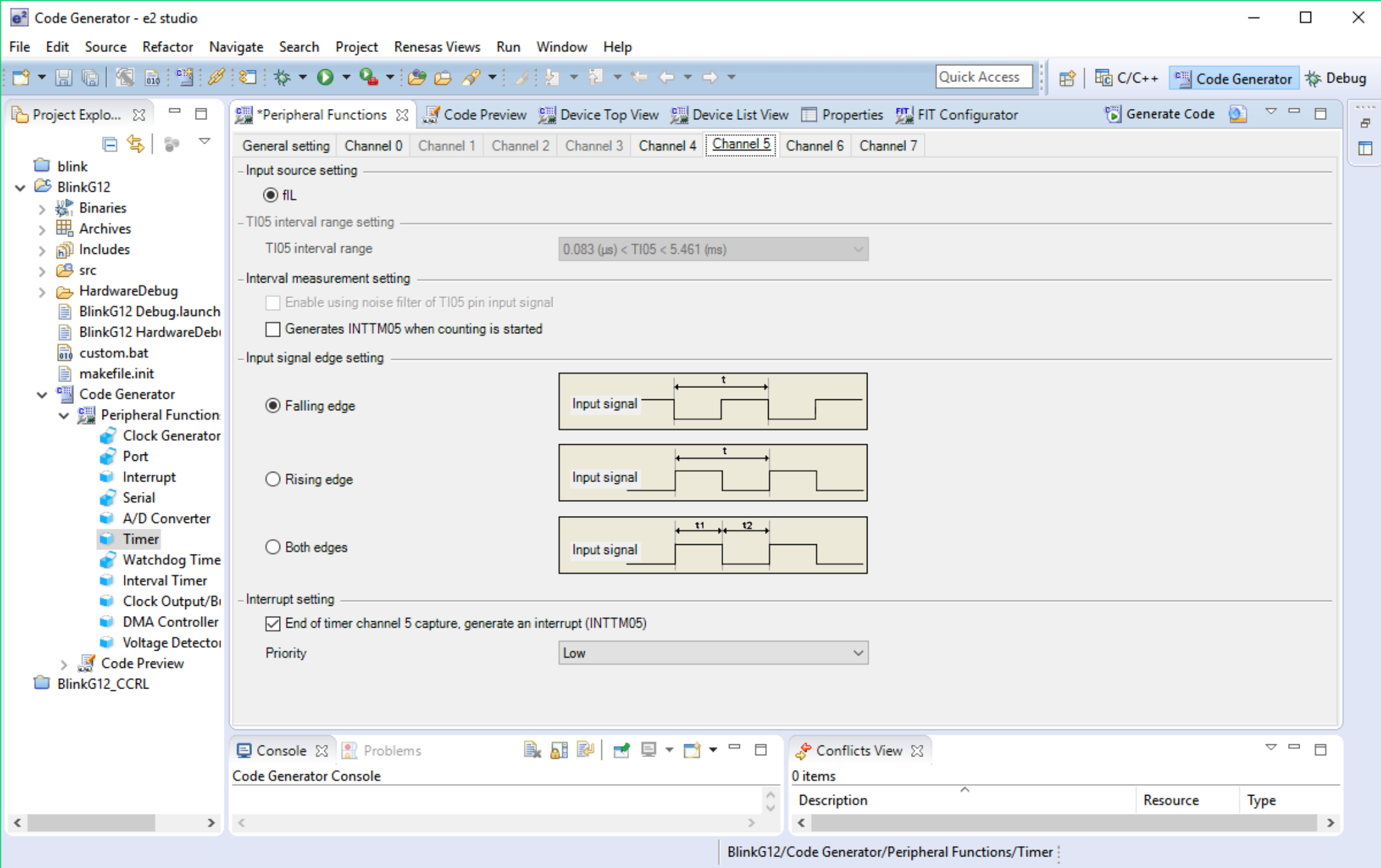
Silicon Labs has a full EFM8 peripheral library, but Simplicity Configurator does not call into it nor include it automatically – it just initializes peripherals, using raw register manipulation.

You can combine Simplicity Configurator projects with the EFM8 peripheral library, or use your own code for interacting with the peripherals during runtime (Or just use the peripheral library without any code configurator).

I really like this approach, as it provides a ton of flexibility, and keeps the size of generated code under control.

It's also one of the snappiest code-gen tools reviewed – Silicon Labs has so much confidence in its tool's speed that it programmed the tool to auto-regenerate all the initialization code whenever you save the Configurator document, and in my testing, this didn't introduce any noticeable lag.

A neat perk unique to Simplicity Configurator is it can also generate stubs for ISRs – and the generated stubs include comments instructing you which bits to clear before leaving the ISR. Beautiful.



I really enjoyed the timer documentation in the datasheet, and I was glad to see much of it made it into the code generator tool. Too many embedded programmers don't understand all the powerful functions of advanced timers, so this tool should help out a lot.

RENESAS CODE CONFIGURATOR

Renesas e2studio includes Code Configurator, their code-gen tool built into their IDE. This tool integrates a nicely-organized visual layout that's extremely readable and self-documenting.

This tool handles communication callbacks particularly effectively – the code generator creates an interrupt that handles the underlying ISR details, but then directly invokes a statically-declared callback function that's stubbed out conveniently for the user. This callback has direct access to the buffer associated with the data, so there's no expensive copying/buffering operation required in Processor Expert, Microchip Code Configurator, or Atmel START.

Unfortunately, some of the other peripherals – especially the timers – focus heavily on initialization code, and don't provide high-level APIs. For example, with the PWM peripheral, you'll still need to directly modify timer registers, which involves looking up the register names in the datasheet, and learning about the values those registers can take.

I would have preferred Code Configurator to generate PWM `setDutyCycle()`-type functions – more advanced users could directly modify the registers instead of relying on those functions, which would then be optimized out of the final code anyway.

NUVOTON NUTOOL PINCONFIG

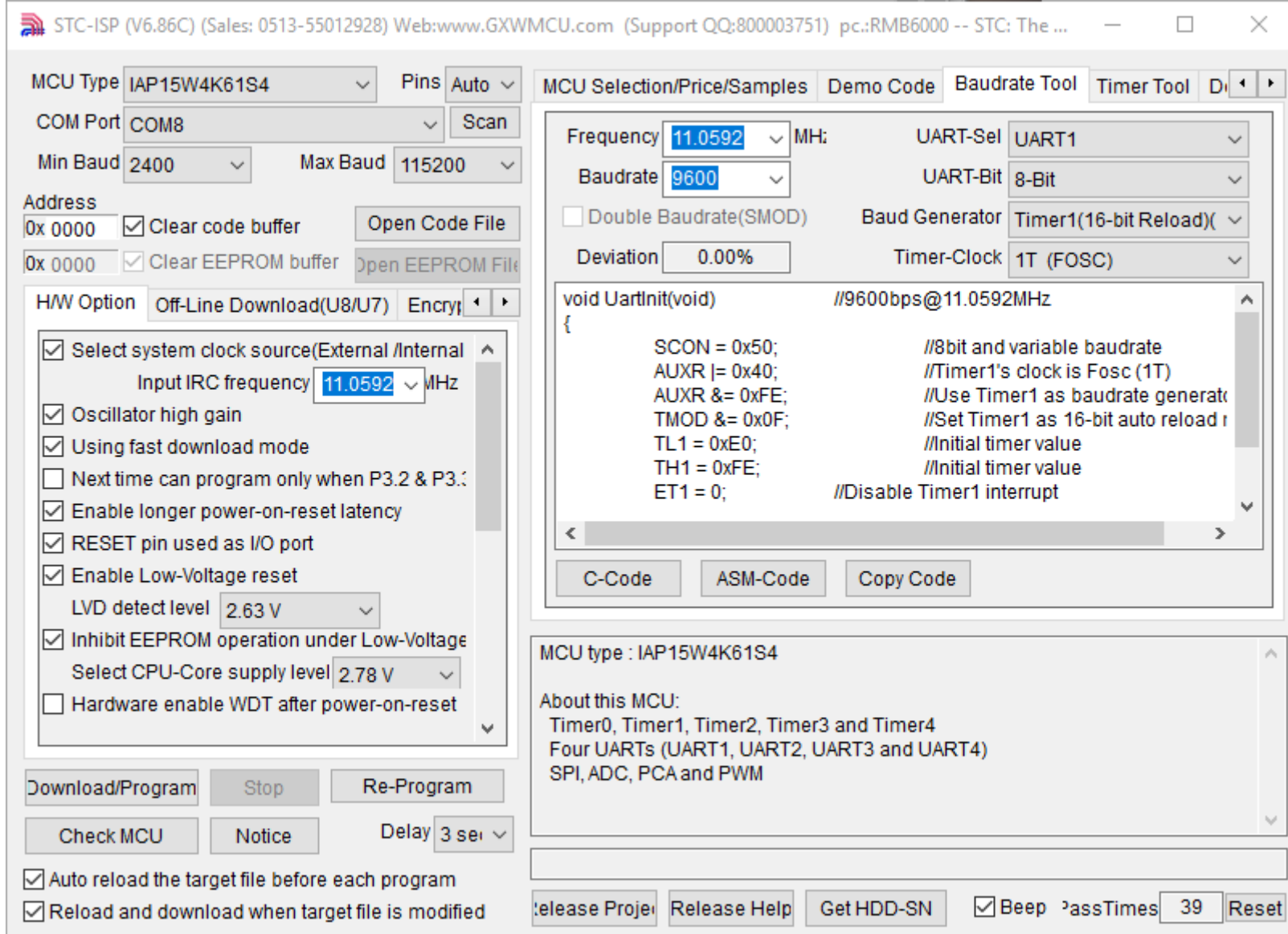
Nuvoton provided a code configurator tool for their M0, but I mention it here only to dissuade you from thinking it has absolutely any value at all, in case you see an advertisement for it. Rather, it is full of bugs, only uses an outdated, buggy peripheral library, and cannot restore configurations properly. Whoever wrote it on their lunch break needs to spend another day or two getting it working before they release version 2.0.

NXP MCUXPRESSO CONFIG TOOLS

Similarly, NXP really dropped the ball on the MCUXpresso Config Tools program, which it uses in lieu of Processor Expert. This stand-alone program purports to do clock configuration and pin-muxing, but in practice, the only code it generates is clock gating code. It claims to generate MCUXpresso projects, but then it informs you it cannot generate projects for MCUXpresso.

You can export a weird archive format that you can import back into MCUXpresso, but if you want to make any changes, you have to go through the whole process again – recreating your project. It's terrible.

I ended up forcing it to generate code in a “generated” folder inside an existing MCUXpresso project. I set up the include paths to look in the new, generated tree structure, and was able to get it going – but this is not a procedure for novices who may be unfamiliar with Eclipse C/C++ development.



STC-ISP is STC's do-everything programmer, parametric search engine, and code-gen tool. It may look a little silly, but it's both easy to use and very functional.

STCMICRO STC-ISP

STC-ISP – STC’s do-everything programmer utility – has some built-in code-generation capabilities. On the 8051, the three things people struggle with the most is UART, timer, and delay functions, as these all require clock calculations dependent on your particular set-up. STC-ISP generates precisely three types of source code: UART initialization, timer initialization, and delay functions. Oh, and it will output both C and ASM code, and it nicely comments the values so you remember why they’re in your source code in 6 months when you go to look at it again.

MPLAB X IDE v3.65 - dmx : default

File Edit View Navigate Source Refactor Production Debug Team Tools Window Help

default PC: 0x0 z dc c : W:0x0 : bank 0 How do I? Keyword(s)

Proj... Files Classes Servi... Res... x MPLAB® Code Configurator x

Project Resources

System

- Interrupt Module
- Pin Module
- System Module

Peripherals

- TMR2
- CCP1
- EUSART**
- CCP3
- CCP2

Device Resources

- CRC
- CWG
- Clock Reference
- Comparator
- DAC
- DSM
- Ext_Interrupt
- FVR
- MSSP
- Memory
- NCO
- PWM
- SMT
- Timer
- ZCD
- Libraries
 - Bootloader Generator
 - LIN
 - mTouch

EUSART

Easy Setup Registers Notifications(Filtered) : 0

Hardware Settings

Mode asynchronous

Enable EUSART Baud Rate: 250000 Error: 0.000 %
 Enable Transmit Transmission Bits: 8-bit
 Enable Wake-up Reception Bits: 8-bit
 Auto-Baud Detection Clock Polarity: Non-Inverted
 Enable Address Detect Enable Continuous Receive

Enable EUSART Interrupts

Software Settings

Redirect STUDIO to USART

Software Transmit Buffer Size 8

Software Receive Buffer Size 8

Pin Manager: Package [MCC]

MCLR 1 28 RB7
 RA0 2 27 RB6
 RA1 3 26 RB5
 RA2 4 25 RB4
 RA3 5 24 RB3
 RA4 6 23 RB2
 RA5 7 22 RB1
 VSS 8 21 RB0
 RA7 9 20 VDD
 RA6 10 19 VSS
 RC0 11 18 RC7|RX
 RC1 12 17 RC6|TX
 RC2 13 16 RC5
 RC3|T2IN 14 15 RC4

Trace - dmx Notifications Output Pin Manager: Grid [MCC]

Package:	SOIC28	Pin No:	2	3	4	5	6	7	10	9	21	22	23	24	25	26	27	28	11	12	13	14	15	16	17	18	1
			Port A							Port B							Port C							E			
Module	Function	Direction	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	3
CCP1	CCP1	output									🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	
CCP2	CCP2	output									🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	
CCP3	CCP3	output									🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	
EUSART	RX	input									🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	
	TX	output									🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	
OSC	CLKOUT	output							🔒																		
Pin Module	GPIO	input	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	
	GPIO	output	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒	
RESET	MCLR	input																								🔒	
TMR2	T2IN	input	🔒	🔒	🔒	🔒	🔒	🔒	🔒	🔒											🔒	🔒	🔒	🔒	🔒	🔒	

dmx - Dash... b2 - Navigator Versions [... x]

Versions

- MPLAB® Code Configurator (Plugin) v3.36

Microchip Code Configurator provides a stellar pin-muxing perspective that other manufacturers haven't quite duplicated.

MICROCHIP CODE CONFIGURATOR

Microchip Code Configurator (MCC) works with some (but not nearly all) 8-bit, 16-bit, and 32-bit PIC parts. This tool provides a nice pin manager view that shows you assignment possibilities for all peripherals. Each peripheral has an Easy Setup or a Registers view that lets you interact with the system at a lower level.

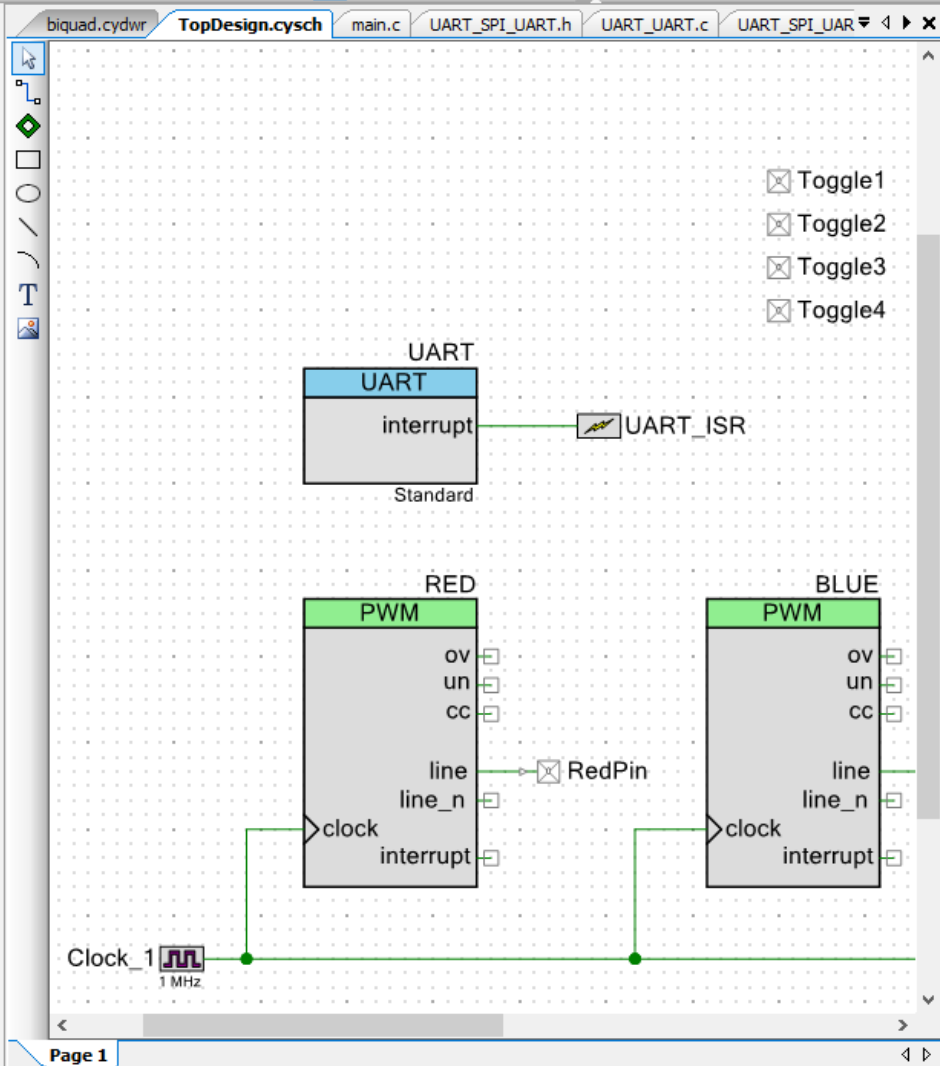
The tool generates a purpose-built, basic, compact runtime library that's a great place to get started with, but I'd like to see the ability to generate ISR stubs – especially for communication peripherals – instead of relying on out-of-ISR-context functions for handling data (which requires buffering data and other overheads).

You can disable automatic reception to prevent MCC from generating these ISRs (allowing you to write your own), but I couldn't find a way of being able to define my own ISR, but also have MCC generate code that automatically sets all the interrupt-enable bits that need to be present for ISRs to execute. Again, this results in an unnecessary trip to the datasheet.

The tool managed to generate extremely compact code – the best in the round-up of code configurator tool (though this is also just a product of the PIC16 architecture), and, like Atmel START's 8-bit code, read like something a human would write.

Workspace Explorer (3 projects)

- *Workspace 'PSOC 4000S' (3 Projects)
 - Project 'biquad' [CY8C4045AZI-S413]
 - TopDesign.cysch
 - Project 'dmx' [CY8C4045AZI-S413]
 - Design Wide Resources (dmx.cydw)
 - Pins
 - Analog
 - Clocks
 - Interrupts
 - System
 - Directives
 - Flash Security
 - Header Files
 - cyapicalbacks.h
 - Source Files
 - main.c
 - Generated_Source
 - PSoC4
 - BLUE
 - BluePin
 - Clock_1
 - cy_boot
 - cy_lfclk
 - GREEN
 - GreenPin
 - RED
 - RedPin
 - Toggle1
 - Toggle2
 - Toggle3
 - Toggle4
 - UART
 - UART.c
 - UART.h
 - UART_BOOT.c
 - UART_BOOT.h
 - UART_PINS.h
 - UART_PM.c
 - UART_PVT.h
 - UART_SPI_UART.c
 - UART_SPI_UART.h
 - UART_SPI_UART_INT.c
 - UART_SPI_UART_PVT.h
 - UART_UART.c
 - UART_UART_BOOT.c



Page 1

Output

Show output from: All

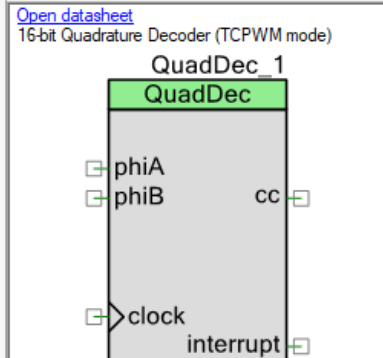
```
Erasing...
Programming of Flash Starting...
Protecting...
Verify Checksum...
Finished Programming
```

Component Catalog (89 components)

Search for...

Cypress Off-Chip

- Cypress Component Catalog
 - Analog
 - CapSense
 - Communications
 - Digital
 - Functions
 - PWM (TCPWM mode) [v2.10]
 - Quadrature Decoder (TCPWM mode) [v2.10]
 - Timer Counter (TCPWM mode) [v2.10]
 - Timer Counter PWM (TCPWM mode) [v2.10]
 - Logic
 - Digital Constant [v1.0]
 - Logic High '1' [v1.0]
 - Logic Low '0' [v1.0]
 - SmartIO [v1.10]
 - Virtual Mux [v1.0]
 - Display
 - Ports and Pins
 - Analog Pin [v2.20]
 - Digital Bidirectional Pin [v2.20]
 - Digital Input Pin [v2.20]
 - Digital Output Pin [v2.20]
 - System
 - Bootloadable [v1.50]
 - Bootloader [v1.50]
 - Clock [v2.20]
 - Global Signal Reference [v2.10]
 - Interrupt [v1.70]



System	
Interrupts	1 / 16
IO	8 / 36
Segment LCD	0 / 1
CapSense	0 / 1
Communication	
Serial Communication (SCB)	1 / 2
Digital	
Timer/Counter/PWM	3 / 5
Smart IO Ports	0 / 2
Analog	
Comparator	0 / 1
LP Comparator	0 / 2
DAC	0 / 2
Memory	
Flash	7.8 %
SRAM	47.9 %

PSoC Creator is built around Cypress's code-gen tools, which provides a schematic capture interface for instantiating and connecting components; these tools generate the configuration bitstream along with an API for the peripherals.

PSOC CREATOR

PSoC Creator is infamous for its deeply-integrated code configurator tool. Because PSoC devices have peripherals whose registers are configured directly from a bitstream in flash memory, the PSoC Creator's tools are not optional.

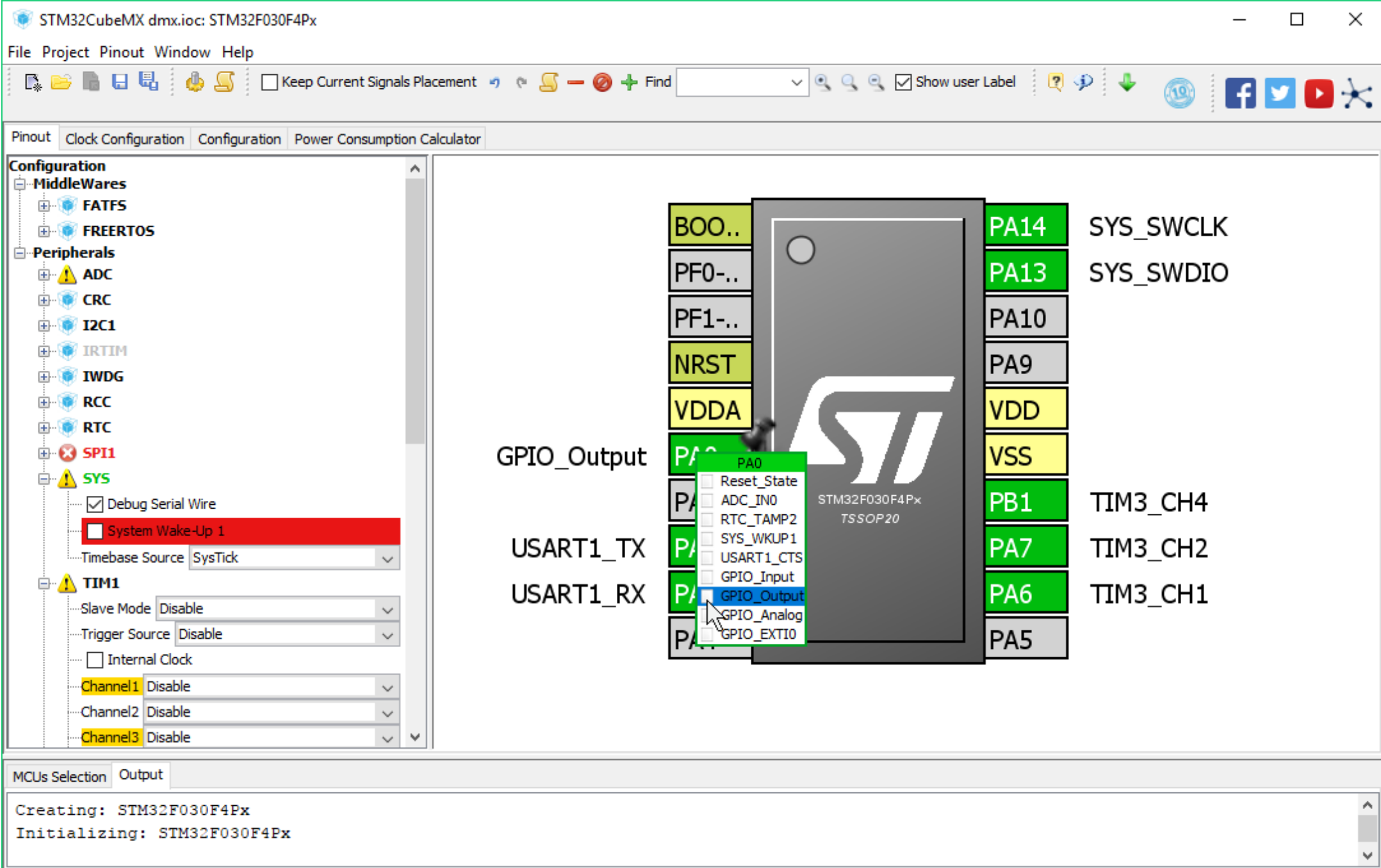
This is an elaborate system that uses a schematic-capture interface to place individual blocks – PWM modules, SPI devices, ADC pins, and – in higher-end PSoC devices – reconfigurable analog and digital blocks.

Each block has a nicely laid-out properties window that allows you to configure nearly everything about the peripheral.

This system seems like it would be popular among people with an electronics – not programming – background. As much of a die-hard PCB layout guy as I am, I actually found the interface to be clunky and unproductive. You have to place *everything* on the schematic – even pins and clock sources. If you want to define an ISR, the “PSoC way” is to wire up an ISR component to an interrupt signal on the schematic.

I suppose you end up with nicely-documented PDF outputs explaining how the components fit together, but again, the system seems clunky and heavy-handed. Do I really have to create and configure a clock component, simply to route it into a timer? Can't the timer just have a clock divider interface built-in?

It's all a matter of personal preference.



STM32CubeMX is a lightweight, stand-alone code-gen tool that focuses on peripheral initialization, while pulling in the standard STM32 peripheral library for runtime duties.

ST STM32CUBEMX

STM32CubeMX (the worst name in my round-up) is a lightweight, stand-alone Java app that's oddly compelling to use. The tool supports initializing almost all peripherals of almost all STM32 devices – an impressive feat on its own. I couldn't find a peripheral mode or feature that the tool didn't support – but it's important to understand the runtime code it generates uses the standard STM32 peripheral libraries, which have bothered some people due to their size.

And, truth be told, these were some of the fattest hex files I made – only topped by Atmel START's D10 code, and Nuvoton's non-LTO'd peripheral library.

The other problem is how basic some of the peripheral drivers are. But because the tool is intelligent enough to ignore strategically-placed user code, it's easy to customize the generated code to suit your purposes.

As a plus, the tool can generate project formats that target all popular development ecosystems – and not just their official System Workbench for STM32.

COMPILERS

There was a large contingent of hobbyists in the 1990s and 2000s who were lured by low-quality proprietary compilers that didn't work well, crashed often, and quickly disappeared from the market without a trace. Those have all but vanished – these days, GCC builds code for several different MCU families, while the industry stand-bys – Keil, IAR, Cosmic, etc – continue to be popular choices for 8-bit MCUs that lack C-friendliness (and thus, GCC support).

The biggest change in the last 10 years is the democratization of tools – even proprietary, expensive compilers tend to have generous code-size limitations (64 KB or more in some cases – plenty for a quick evaluation or hobbyist projects).

And some vendors – like Silicon Labs and ST – work with compiler vendors to offer full versions of these expensive tools free of charge to the customer.

I conducted all my testing using nothing but freely-available versions of the vendor-recommended toolchains for their products. Throughout the months of testing on 21 different parts, I didn't run into a single compiler bug – but I did run into several peculiarities and optimization snafus.

GCC

GCC is an open-source, cross-platform part of the GNU Project, and first released in 1987. Since then, developers have ported GCC to dozens of architectures. In the microcontroller world, it can target the Arm, AVR, RL-78, and MSP-430 parts reviewed – as well as parts I wasn't able to get to, including Xtensa, RX, M16C, SH, NIOS2, and Microblaze.

GCC can support MIPS processors, and there are several projects ([e.g.](#)) from people trying to build code for the PIC32, but you'll get no help from Microchip and a [growing chorus](#) is [questioning](#) whether they're in violation of GPL.

GCC excels at producing fast math code, supporting recent C standards, and – with link-time-optimization – producing compact code.

Arm, especially, has seen a mass migration to GCC – even Arm Holdings (who own MDK-ARM) take part in GCC's upkeep, and [distribute binary releases](#) of it on their site. I know several smaller shops that have stopped renewing their Keil MDK licenses, because GCC is as good – or better – than MDK.

I haven't conducted thorough benchmarking myself, but in my biquad filtering test with the Nuvoton Cortex-M0 part, the GCC implementation required 30 clock cycles, while the MDK version needed 42. Code size was also better for GCC than MDK – 2880 bytes versus 3004 bytes.

The Achilles' heel for GCC on MCUs is actually not GCC itself – but rather the C library – Newlib or Newlib-Nano – that is often linked to the program whenever you need `printf()` or similar C routines.

Here, Keil MDK, IAR, and other vendors can produce much smaller code size, as they're using their own C libraries that are highly-optimized for the architecture (and may not be 100% compatible with C99).

This compares with Newlib, the full GCC C library that's aimed at much larger devices running full operating systems – or Newlib-Nano, which trims down, but is still much larger than proprietary MCU C libraries.

I want to highlight NXP's MCUXpresso IDE, as they're the only vendor to distribute a built-for-ARM C library, [Redlib](#), with their IDE. Redlib is part of their Code Red acquisition. All other vendors copy-and-paste the same newlib and newlib-nano options.

[Raisonance conducted a benchmark](#) a few years ago that confirms everything I've written on the strengths and weaknesses of GCC.⁹ Full disclosure: Raisonance has a stake in GCC, as they use it in their Ride7 proprietary IDE.

One minor annoyance with GCC is that its backend considers SFR accesses as an “optimization” – so with the optimizer off, it uses fairly slow code for accessing peripherals. I've verified this with both AVR and RL-78 targets.

On avr-gcc, compiling with -Og or higher will fix this issue, but this can introduce debugging headaches (though -Og minimizes them).

GCC's RL-78 port, however, seems crippled by this issue – no optimization flags I tried were ever able to generate correct SFR accesses; everything always went through normal 16-bit space, issuing a strong performance penalty.

CC-RL

While the RL-78 has rudimentary support for GCC, as I mentioned above, I was never able to coax it into generating good register I/O. CC-RL, the proprietary RL78 compiler from Renesas, has no issues with this.

But CC-RL doesn't inline non-native math functions – instead, it calls into runtime libraries, which introduces extra overhead.

The free mode supports 64 K linking.

KEIL C51

The 8051 harkens from a time where developers programmed microcomputers (and microcontrollers) in assembly, not C. Its fancy control-friendly features like small sets of banked registers (which can interchange in an interrupt context) don't play well with compilers.

Worst still, the 8051 suffers from a small, 8-bit stack that struggles to keep up with traditional stack-based C implementations.

Early C compilers for the 8051 often started as 68K or x86 compilers hacked with an emulated software stack stored in XRAM. This produced code that dawdled through tasks at a snail's pace.

¹⁰ PL/M-51 PL/M-51 wasn't a C compiler – it actually compiled code written PL/M, a proprietary Intel high-level language. Man, the 80s were weird. was an Intel compiler introduced in 1980 that got around this problem by passing variables in defined RAM locations.

Keil took this idea and ran with it. They introduced C51 in 1988 – and it flourished in popularity.

One of the problems with Keil is its trigger-happy use of nonstandard reserved words. But this what you would expect: when you're building a C compiler for the 8051, you *have to* have the ability to declare variables in XRAM versus RAM – or be able to instruct the compiler to make a function reentrant if necessary. You don't have the luxury of a stack-friendly microcontroller which treats all RAM equal.

While that's unavoidable, the way Keil implemented these special attributes is not. Keil obnoxiously sets aside: alien, bdata, bit, code, compact, data, far, idata, interrupt, large, pdata, reentrant, sbit, sfr, sfr16, small, using, and xdata.

Because, as we all know, no developer ever declares variables named things like "data," so that should work out fine.

This is a serious problem when porting large stacks built for other compilers, like GCC. ¹¹ Yes, I know you can disable these extensions with the NOEXTEND compiler directive... but obviously then you can't use these directives.

I would have preferred GCC-style ¹² __attribute__((xdata)) or at least __xdata.

A bigger problem with C51 is that, like CC-RL, it calls into pre-built math libraries whenever it needs to perform a non-native operation (such as a 16-bit multiply), instead of inlining the appropriate set of operations, as GCC does. No optimization setting will fix this behavior, and because function calls are expensive on the 8051, this has dramatic performance implications.

MICROCHIP XC

Microchip produces microcontrollers of three basic designs: an 8-bit, a 16-bit, and a 32-bit. The XC8, XC16, and XC32 are the current compilers in their collection that target each of these, respectively.

These are quite different processors and, under the hood, these are quite different compilers.

The 8-bit processors, in particular, have gone through slow, incremental changes over the years: from 12-bit, to 14-bit, to 16-bit program word sizes, with each bump adding more address space, new instructions, and a bit more stack space.

With each improvement, these PIC devices became easier to write compilers for. The PIC18 was the first core for which Microchip supported C programming – though Hi-Tech had already developed a Keil C51-like compiler for the lower-end devices that lacked a proper stack.

Eventually, Microchip acquired Hi-Tech, and combined these two disparate products into XC8, which covers all 8-bit PIC devices.

None of the 8-bit PIC parts have a usable stack to store variables. To handle this shortcoming, XC8 can create a software stack on PIC18 devices, which have enough indirect addressing operators to support this.

Not so on PIC10/12/16 devices. Like Keil C51, XC8 will reuse certain RAM addresses to hold local auto variables and function parameters. This works well, and is often more efficient than a stack-based approach simpler compilers can use on more advanced hardware. As mentioned with Keil C51 above, the big problem is reentrancy – when a function attempts to call itself (i.e., recursion), or when an ISR calls the same function it happened to interrupt. Here, these auto variables – which should be unique to the function's execution – will be at the same address, thus causing possible corruption.

XC8 beats out Keil C51 by tacitly duplicating any function that ISRs call into, eliminating the ISR reentrancy problem. Unfortunately, unlike Keil, XC8 has no way of forcing the compiler to generate reentrant-capable code on these midrange devices.

They support optimization through the “Omniscient Code Generation” optimizer. According to the documentation, for well-written code, the main advantage to OCG is bank-tracking. Because the PIC devices use banked memory pages (for both RAM and SFRs), most source code literals bank-select statements everywhere.

Most of the other optimizations – removal of dead code, unused variables, unreachable code, unused return expressions, and redundant assignments – seem only relevant to poorly-maintained code.

XC16 and XC32 use GCC. Microchip doesn't advertise the source code, but it's available from their [Archives page](#).

```

* | [15:8] | DLY      | TX Delay time value
* |       |          | This field is use to programming the transfer delay time between the last stop bit leaving the TX-FIFO
* |       |          | and the de-assertion of by setting UA_TOR. DLY register.
*/
_IO uint32_t TOR;

/**
 * BAUD
 * =====
 * Offset: 0x24  UART Baud Rate Divisor Register
 * =====
 * |Bits   |Field      |Descriptions
 * | :----: | :----:    | :----:
 * | [7:0]  | BRD_LowByte|Baud Rate Divider
 * |       |           | The low byte of the baud rate divider
 * | [15:8] | BRD_HighByte|Baud Rate Divider
 * |       |           | The high byte of the baud rate divider
 * | [27:24] | Divider_X | Divider X
 * |       |           | The baud rate divider  $M = X+1$ .
 * | [28]   | DIV_X_ONE | Divider X equal 1
 * |       |           | 0 = Divider  $M = X$  (the equation of  $M = X+1$ , but Divider_X[27:24] must > 8)
 * |       |           | 1 = Divider  $M = 1$  (the equation of  $M = 1$ , but BRD[15:0] must > 3).
 * | [29]   | DIV_X_EN  | Divider X Enable
 * |       |           | The BRD = Baud Rate Divider, and the baud rate equation is
 * |       |           | Baud Rate = Clock / [ M * (BRD + 2) ] ; The default value of M is 16.
 * |       |           | 0 = Disable divider X (the equation of M = 16)
 * |       |           | 1 = Enable divider X (the equation of M = X+1, but Divider_X[27:24 must > 8).
 * |       |           | NOTE: When in IrDA mode, this bit must disable.
 */
_IO uint32_t BAUD;

/**
 * IRCR
 * =====
 * Offset: 0x28  UART IrDA Control Register
 * =====
 * |Bits   |Field      |Descriptions
 * | :----: | :----:    | :----:

```

Nuvoton provides an entire transcript of the reference manual attached to each symbol in their header file, which is a huge step above other header-based documentation.

HEADER FILES

I didn't realize how important good header files were until I started working with some of these parts that simply have horrendous headers.

Since MCUs rely on setting, clearing, toggling, and inspecting bits in registers, it's convenient for the compiler and header files to provide methods for setting and clearing individual bits (and preferably bit ranges) inside registers – even if the architecture doesn't support atomic bit instructions.

Bit-Addressable Register Access

All Microchip PIC parts, Atmel [SAM D10](#), and the [RL-78](#) all support bit-addressable register access. The SAM D10, in particular, is the only Arm part I've ever seen this feature in – and it's fantastically useful – even if their notation is a bit heavy-handed. I'd love to see them go through and #define flat symbols to point to those structs, so instead of:

```
TCC0->CTRLA.bit.ENABLE = true;
```

I could write

```
TCC0_CTRLA_ENABLE = true;
```

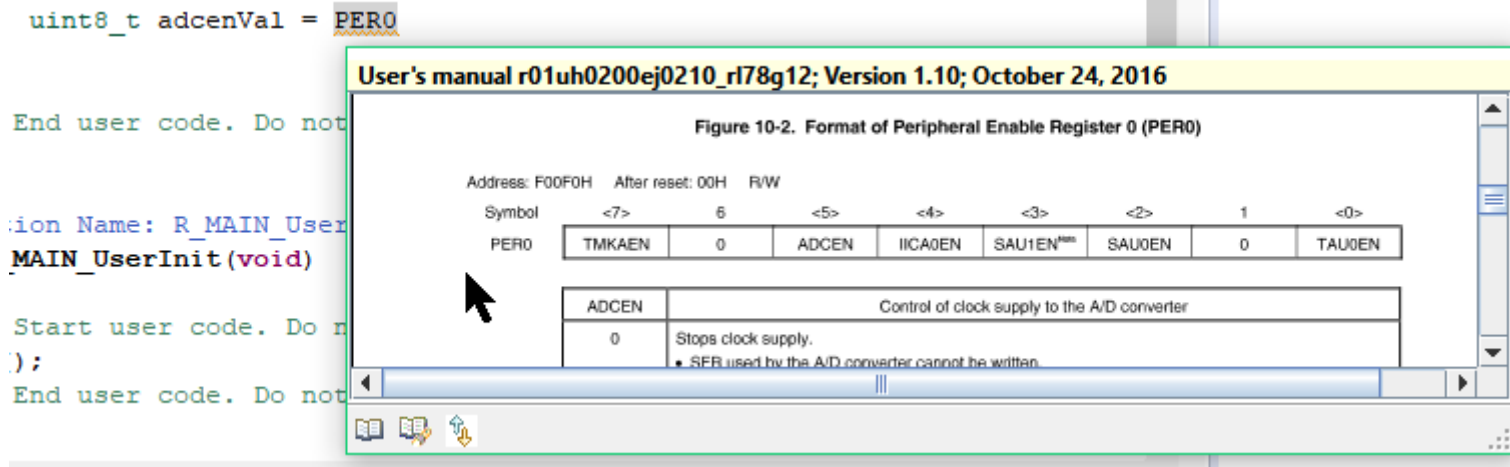
But at this point, I'm just nit-picking.

While Renesas provides bit-addressable register definitions, there are absolutely zero docs in the header files. Same goes for Microchip PIC – there's next to no documentation in the header files for the [PIC16](#), [PIC24](#), and [PIC32](#).

It's 2017 – storage is basically free. Header files should come chock-full of documentation, so you can keep your focus on your code – instead of having to jump around inside PDFs.

Header Documentation

By far, the best-documented header files were from the [Nuvoton M051](#). Essentially they copy-and-pasted the entire reference manual into their header files. You get every register name and description, with every bit described in the same detail as in the official documentation. This is absolutely incredible, and really sets the standard in the industry.



Renesas, while not having [RL-78](#) header documentation, does have a bizarre pop-up PDF viewer thing that displays the relevant reference manual section when you hover-over a variable. I kind of like it, but it's pretty heavy, and I think if I used the architecture more, I'd prefer the Nuvoton approach.

Other than that, [Infineon](#), [ST](#), [NXP](#), and [Texas Instruments](#) provided good documentation in their header files – though I have to fault Eclipse for not displaying this more prominently in tooltips. Often, I had to click-through to the device header file to read the register description. Atmel Studio handled doc display much better.

[Silicon Labs](#) also provided excellent documentation, and along with the other 8051s, had all bit-addressable registers defined clearly. But because of the 8051 architecture, not all registers are bit-addressable – in fact, most aren't.

I wish Keil C51 could be coaxed into generating ORL and ANDL bitwise instructions from some sort of proprietary bit definition structure that would allow **all** bits to be manipulated independently in the same way.

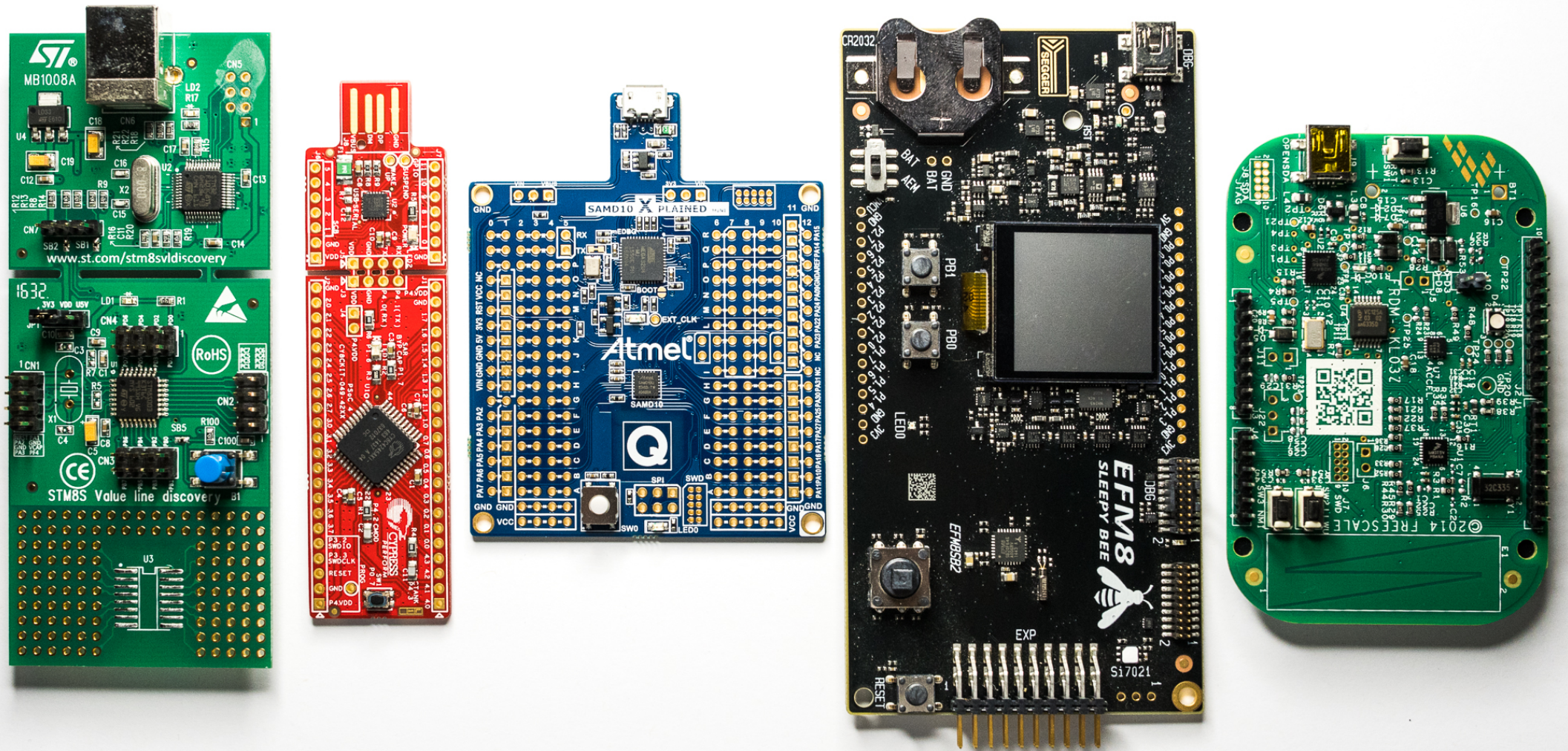
Even more puzzling is why there are no bitfield register definitions for the [tinyAVR](#) or [megaAVR](#), even though the architecture clearly supports bit manipulation. Surely there could be a proprietary AVR-GCC attribute that allowed Atmel to build header files that the compiler could use to generate `sbit` and `clrbit` instructions, right?

Predefined Offset Bitmasks

A (slightly inferior) alternative to bit-level register access is predefined offset macros. This is the route Silicon Labs, Texas Instruments, Infineon, NXP, and Microchip go (though again, the first three provide much better documentation). Atmel also provides this – but only for their `tinyAVR`.

The Worst Headers

The worst header files were from the `megaAVR`, the [PSoC 4000S](#), the [Kinetis KE04](#), the HT-66, the [Sanyo LC-87](#). These header files have zero documentation, no predefined bit offsets, and no bit-addressable register definitions. Their header files are little more than register names attached to addresses.



From left, the STM32, PSoC 4000S, Microchip SAM D10, Silicon Labs EFM8, and NXP Kinetis KL03 development boards.

DEVELOPMENT TOOLS

Development tools have been shrinking and simplifying over the last few years; no longer should you plan on spending hundreds of dollars on a giant beast that combines every peripheral imaginable into a horrible mess of poorly-documented schematics.

For general-purpose projects that these MCUs are geared toward, I still often breadboard a first iteration of the system (usually with break-out boards for the MCU, plus modules for the peripherals). If I'm working with a big microcontroller that has a lot of support circuitry, I'll typically use an off-the-shelf dev board. For smaller parts, I'll toss them on a breakout board.

Breakout-Friendly Dev Boards

In terms of overall form factor, I'll allow my personal preferences to gush out: I *love* dev boards that integrate a debugger and a microcontroller, with nothing more than break-out pins. I think a dev board should have a row of jumpers that allow you to completely disconnect the target from the debugger.

Regardless of what marketing departments think, dev boards should be free of extraneous sensors, buttons, LEDs, or anything else that a user can easily breadboard. I'm OK with capacitive-touch sliders, as long as there isn't a lot of junk (passives) hanging off the lines.

Consequently, I really enjoyed the MSP430 (and all Texas Instruments) LaunchPad boards. The integrated power measurement functionality is wonderful, and I love the no-frills jumper arrangement used to disconnect the target from the on-board debugger.

I didn't get a chance to play with the Microchip Curiosity boards, but these look appealing as well.

Snap-Apart

The snap-apart dev boards were also fun to play with – the Nuvoton N76 and M051 boards, the STM32Discovery, and the Infineon XMC1100 board. Having said that, you think they're going to be great until you try to put them back together again. I spent several minutes in front of my bench grinder cleaning up all the snap-apart boards so they could accept standard 0.1" headers and jumpers.

Arduino UNO Form-Factor

One pet peeve of mine is the insistence on building Arduino Uno form-factor dev boards.

I'm totally cool with vendors building UNO-compatible boards that support Wiring programming. This is a great way to get beginners interested in moving beyond Arduino. Nuvoton does a good job of this with the [M0 series](#). Their [NuMaker Uno](#) board uses a 5V-compatible Arm Cortex-M0 part, and they [provide a board support package](#) for the Arduino IDE.

But a lot of vendors build boards that are cosmetically similar to Arduino UNO dev boards, but have none of the software support necessary to use them with Arduino libraries or the Wiring environment in general.

While many vendors are guilty of this, I have to chastise NXP specifically for this.

NXP used to build fantastic little dev boards – the [LPCXpresso “stick” boards](#) – that were the first in the industry to have snap-off (well, cut-off) debuggers. They broke out all the pins of the MCU onto sensible 0.1” headers, making breadboard prototyping easy. These boards were great.

Unfortunately, they've discontinued them and switched to their new [LPCXpresso MAX](#) boards. These new boards cram their processors – which are very unlike the ATmega328p – into an Arduino Uno form-factor.

This is especially stupid for parts like the [LPC81x series](#), which only have 11 GPIO pins (excluding SWD and crystal) and don't even have an ADC.

So what does NXP do to fit their square peg into a round hole? They throw in random I²C GPIO and ADC peripherals to pad the LPC's I/O count (and endow it with an ADC it doesn't inherently have) – and then provide zero software on their downloads page for actually interacting with these peripherals.

And in targeting this form factor, NXP hilariously misses the point. Do they provide Wiring support for these boards to be used inside Arduino? Nope. Do they provide libraries for all the Arduino shields that are supported by their MAX boards? Nope. Do they even have a list of which shields are compatible, and which ones are incompatible with various boards? No way.

So, they've saved me 30 seconds by allowing me to plug a shield directly into this board (without having to use jumper wires), but haven't done any of the actual work for me.

At the same time, their debugger is no longer separable from their target, there's a boat-load of random crap on the board that must be removed for accurate power consumption figures, and the pins no longer line up with standard 0.1” breadboards.

And even if it *were* properly executed, I think I reject the underlying goal of this strategy. The *whole purpose* of using an Arduino UNO form-factor is to *support Arduino shields* – yet I have *never* seen an Arduino shield on the desk of a professional engineer or advanced hobbyist. At least, not one plugged into one of these “Uno form-factor” dev boards.

Dev Board Surgery

For the low-power part of this project, I had to do some minor surgery to the STC8, Nuvoton, and the three Atmel Xplained Mini boards covering the tinyAVR, megaAVR, and SAMD10. This mostly involved removing LEDs or separating a power trace. The schematics were clearly documented, and I consider these changes to be completely routine – just the price of admission of trying to do low-power development.

While the Cypress PSoC 4000S dev board looks uncomplicated, I struggled to get accurate power consumption without several hacks that weren’t clearly documented.

The Infineon board’s power subsystem was slightly confusing, as there’s no power going between the debugger and the rest of the board once you snap the two apart. Once I figured that out, I had no further issues, and I like that they took the time to put headers on the board in a way that allows the two parts to be re-joined together.

I hate to say this because I was a huge fan when they first came out, but I’ve grown to fundamentally hate the Freescale (now NXP) FRDM dev boards. To even get my project to *work*, I had to do *major* reworking on the Kinetis KL03 FRDM board, which contained random filtering capacitors on UART RX pins. These weren’t mentioned in the documentation and were only traceable by studying six pages of schematics that came with the board.

Trying to get accurate low-power measurements was a further struggle on the FRDM board, which is full of button pull-ups, LEDs, I2C sensors, and no good power separation capabilities. This dev board is a total disaster – I wouldn’t recommend it to my worst enemy. NXP needs to take a step back, simplify the board to its essence, and try again.

I also had to do major hacking of the Renesas RL-78 promotional board, but this was because I was trying to be a cheap-skate and convert the kit to a general-purpose RL-78 programmer (since I needed to target a completely different microcontroller) – I can’t fault Renesas for that, but I *do* wish their boards had debugger jumpers that would have kept me away from the soldering iron.

On-Board Debugging & Other Features

All dev boards tested have built-in on-board debuggers, but they varied widely in capabilities.

In terms of functionality, the best dev board in the round-up is the [Silicon Labs SLSTK boards](#) have a full, USB 2.0 high-speed [J-Link On-Board](#) debugger that has been augmented with a high-dynamic-range energy monitor, which is an amazing deal considering the cost of the dev boards (\$30). While I only reviewed the EFM8 series in this review, the EFM32 ARM processors they make have nearly identical development boards.

The MSP430 LaunchPad also has an energy monitor, which is still a rare feature to find in this pricing.

While other manufacturers have J-Link OB debuggers, they're usually USB 2.0 full speed, not high speed. This is the case with the Freescale KL03 and Infineon XMC1100 boards.

Cypress and Microchip's SAM D10 went with CMSIS-DAP debuggers on their boards, while the Nuvoton used their proprietary NuLink for both their Arm and 8051 controllers.

Both Freescale and ST enable reprogramming the debugger firmware on their dev kits with SEGGER J-Link firmware. I don't think anyone in the industry disagrees that J-Link is, by far, the best Arm debugger – and even operating at USB 2.0 full-speed specs (12 Mbps), flash download is snappy, and stepping through code is a breeze.

DEBUG ADAPTERS

While you can hack all current development boards to provide off-board debugging, at best this is clunky, and at worst it can be a violation of the EULA for the dev board. Serious hobbyists and professionals usually have dedicated debug adapters laying around that can easily plug into different targets they're designing.

In the Microchip camp, your choices are the [PicKit3](#) (\$48), [ICD4](#) (\$249), and [RealICE](#) (\$499) – all of these debug adapters will work with all current Microchip PIC parts, but the PicKit3 is substantially slower than the ICD4, and only supports two breakpoints.

The Microchip (née Atmel) AVR and SAM parts are a different story: the \$130 [Atmel-ICE](#) is your only option, assuming you want a debugger that can target all current Atmel parts. If you're only interested in debugging older parts, like the megaAVR and non-1-Series tinyAVR parts,

you can get by with a [\\$49 AVR Dragon](#).

The Silicon Labs EFM8 ecosystem has some of the fastest debug times, as well as some of the lowest-cost debugger hardware. Price descending, your options include the [SEGGER J-Link](#) debugger (\$60-1000), the [USB Debug Adapter](#) (\$35), the [ToolStick Base Adapter](#) (\$19), semi-sanctioned [eBay clone adapters](#) (\$10), or the [bare debugger chip](#) you can use in your own design ([\\$1.78](#)).

In the Arm ecosystem, just shut up and buy a J-Link. Seriously. It works in every Arm IDE, with every Arm part on the market. It has the fastest debug speeds, supports any target voltage, and has unlimited software breakpoints. If you're a student, you can get a [J-Link EDU Mini for \\$18](#) (cheaper than clones), or the [full EDU version for \\$60](#). If you're a professional, buy the [\\$600 commercial version](#) – it's worth the handful of billable hours you'll have to charge to pay for it.

Debug adapters varied widely in speed – with the PIC16, PIC32MM, and SAM D10 taking the longest to program flash memory – the latter two took 20 seconds to program their entire flash, which is entirely too long.

These parts also had the most inconsistent speeds – sometimes if I restarted my computer, they would be significantly slower or faster. I didn't spend time fully characterizing their ineptitude, though, so your mileage may vary.

The fastest IDE flash load times came from the Infineon XMC1100, running the J-Link firmware, which could fill its entire 8 KB of flash and run to main() in 2.47 seconds. That's impressive, coming from an Eclipse-based IDE not known for its debugging kick-off abilities.

Actually, the Sanyo LC87 beat it out at 1.87 seconds, but this is using their special RD87 application, which requires jumping away from their IDE to use (and has a ton of manual steps involved in loading the flash file), thus I'd take this result with a grain of salt.

Other than that, the EFM8, STM8, and STC8 all had sub-5-second debug speeds when loading average-sized programs, and none took more than 6.2 seconds to fill their flash memory.

PERFORMANCE

BIT TOGGING

ATMEL TINYAVR

4 CYCLES

ATMEL MEGA AVR

3 CYCLES

ATMEL SAM D10

3 CYCLES

CYPRESS PSOC 4000S

11 CYCLES

FREESCALE KE04

3 CYCLES

FREESCALE KL03

4 CYCLES

HOLTEK HT66

16 CYCLES

INFINEON XMC1100	9 CYCLES
MICROCHIP PIC16	20 CYCLES
MICROCHIP PIC24	10 CYCLES
MICROCHIP PIC32MM	3 CYCLES
NUVOTON N76	7 CYCLES
NUVOTON M051	8 CYCLES
NXP LPC811	4 CYCLES
RENESAS RL78	5 CYCLES
SANYO LC87	18 CYCLES
SILICON LABS EFM8	8 CYCLES
ST STM8	4 CYCLES
ST STM32F0	9 CYCLES
STC STC8	4 CYCLES
TI MSP430FR	7 CYCLES

Some Cortex-M0+, megaAVR, and PIC32MM were able to hit 3 cycles with the help of single-cycle GPIO toggling and two-cycle jump instructions.

Cortex-M0 parts can theoretically hit 5 cycles (which the XMC1100 did when running from RAM), but many of the Arm microcontrollers tested need much more – as much as 11 – due to flash caching strategies or the lack of GPIO toggle registers.

Flash reading also plagued the [EFM8 Laser Bee](#), which takes 8 cycles to unconditionally jump when operating at 72 MHz (though, oddly, its 8-cycle toggling performance is better than the 10 cycles it should take, if one trusts the datasheet). The [N76](#) turned in poor results caused by poor bit math performance, while the [STC8](#)'s 4-cycle toggle was right up there with the best.

The [tinyAVR](#) is a bit slower than the [megaAVR](#) due to its 16-bit peripheral address space, but there are special provisions for remapping specific GPIO ports into the 64-byte address space, which will give it identical performance (at the expense of only being able to access a single GPIO port from this mode).

At the bottom of the pack are the 4T and 3T architectures – the [Holtek HT66](#), the [Microchip PIC16](#), and the [Sanyo LC87](#). These 8-bit architectures must essentially load the working register with the toggle bit, XOR it into the port latches, and then jump back to the top. I'm not sure why the working register has to be reloaded each time (does the result of the XOR operation end up back in the working register before being written to the latch outputs?), and I'm planning on investigating some strange bank-select code that XC8 injects in the PIC16 loop.

Speaking of weird Microchip compiler issues, XC16 – even with the optimizations cranked all the way up – generates 5-machine-cycle bit-wiggling code for the [Microchip PIC24](#), instead of a "btg" followed by a jump (which should be three machine cycles – 6 clock cycles – total).

BIQUAD FILTERING

FILTERING SPEED	CLOCK CYCLES	POWER CONSUMPTION	EFFICIENCY

ATMEL TINYAVR	159.80 KSPS
ATMEL MEGAAVR	123.27 KSPS
ATMEL SAM D10	1822.32 KSPS
CYPRESS PSOC 4000S	885.69 KSPS
FREESCALE KE04	1715.36 KSPS
FREESCALE KL03	1645.24 KSPS
HOLTEK HT66	2.71 KSPS
INFINEON XMC1100	805.84 KSPS
MICROCHIP PIC16	21.83 KSPS
MICROCHIP PIC24	838.46 KSPS
MICROCHIP PIC32MM	829.88 KSPS
NUVOTON N76	38.79 KSPS
NUVOTON M051	1732.07 KSPS
NXP LPC811	731.93 KSPS
RENESAS RL78	731.68 KSPS

SANYO LC87

23.55 KSPS

SILICON LABS EFM8

202.40 KSPS

ST STM8

79.49 KSPS

ST STM32F0

1647.79 KSPS

STC STC8

156.56 KSPS

TI MSP430FR

129.95 KSPS

I'm going to discuss the 8-bit and 16/32-bit results separately, as there are nearly three orders of magnitude of variation among all the parts – with a clear demarcation between the 8-bit and 16/32-bit parts.

16/32-BIT PROCESSORS

Except for the [MSP430](#) – which doesn't have a hardware multiplier in the part I tested – all the 16- and 32-bit microcontrollers had similar results in terms of clock-cycle efficiency.

The Arm parts were the fastest in the round-up. I think the [Atmel D10](#)'s PLL was running a bit hotter than 48 MHz, which allowed it to edge out over the other 48 MHz ARM parts.

Looking at clock efficiency, I suspect the variation we see between parts is primarily the result of flash caching. Parts that have good flash caching can get near 27 clock cycles per filtering loop.

The [PSoC](#) doesn't need a flash accelerator since it can only run at 24 MHz – thus, it can access flash at full speed with no wait-states, and can consequently hit 27 cycles.

The [Infineon XMC1100](#), on the other hand, only seems to be able to read flash with no wait-states when operating at 8 MHz or below. It has no flash accelerator either, which deeply penalizes its scores. However, when the part runs code from RAM instead of flash, it hits 27 cycles (at very low power figures, too). I discuss this more in the [XMC1100 article](#).

The MIPS-based 32-bit [Microchip PIC32](#) holds its own – just three cycles shy of the Cortex-M0+ results. Like the [PSoC 4000S](#), this part runs slow enough to not need a flash accelerator. Before committing to higher-speed PIC32MX devices, I'd have to investigate the flash caching.

I was blown away by the performance of the 16-bit [Renesas RL-78](#), which only has 8-bit data pathways. It was only 6 cycles away from the hot-rod Cortex-M0+ parts, and pulled 7 cycles faster than the [PIC24](#), which is a true 16-bit design. I only wish it had a faster core speed.

8-BIT PROCESSORS

Compared to the 16/32-bit results, the 8-bit processors can be described as either “mediocre” or “absolutely awful.”

The [Microchip PIC16](#) and [Holtek HT-66](#) – with similar single-register 4T architectures – arrive at the party dead-last in terms of clock cycle count. These parts simply aren't built to be doing much math at all; let alone 16-bit multiplies and accumulates.

Because slow microcontrollers take a long time to process data, they're often overwhelmed by static power consumption, and this test verifies this. The HT66, LC87, and PIC16 both required more than 500 nJ/sample to process the data – so much that I ended up clipping the data in the graph to help better show the variation of the more-efficient parts (without resorting to logarithmic plots).

At the other end of the spectrum lies the [tinyAVR](#) and [megaAVR](#) processors, which both brought in excellent clock-cycle counts.

By the way, it's important to note that neither of these AVR parts are running at full speed in this test. The [tinyAVR](#) has an internal 20 MHz oscillator that has to be scaled down to 10 MHz so I can run the part at 3.3V without violating the ratings in the datasheet. The [megaAVR](#) only has an 8 MHz internal oscillator, so that's what I have to run it on.

Both these parts are capable of 20 MHz operation, so it's a little silly to have to down-clock them because Atmel doesn't build these parts on a modern process allowing 1.8-3.6V operation at full speed, like most other parts tested. Every other part tested has a full-speed, better-than-2% oscillator; the [megaAVR](#) is stuck with an atrociously inaccurate 8 MHz job that severely limits the performance you can get out of this part. It's 2017 – crystals on MCUs should only be necessary for RTCs and RF.

Anyway, back to the data: this should settle the PIC vs AVR debate about performance. Yes, the AVR typically has a lower clock speed, but unless you can run a PIC16 at 176 MHz, an 8 MHz AVR is going to win the math performance tests handily.

The [Sanyo LC87](#) and [STM8](#) are both better than the [HT66](#) and the [PIC16](#), but both have slow core speeds that limit their performance.

BATTLE OF THE 8051S

One of the most interesting narrative that came out of this test was the wildly different performance numbers the 8051s produced.

First thing's first: Keil C51 struggled to generate good code in the biquad experiment – the biggest problem being the 16-bit multiplication. Rather than producing raw assembly that operates on whichever registers end up with these variables, Keil generates function calls into a signed-16-bit multiply library routine. This has drastic performance implications when compared to the much-better AVR-GCC code.

With that said, the three different 8051s running **identical** binary images produced very different results – caused by both clock-cycle efficiency and core speed. Let's dig in a bit:

STC8
EFM8
STC15W
Nuvoton N76

I've plotted a cycle-count cumulative distribution above – I've included the STC15, an older part closely related to the [STC8](#) (and mentioned extensively in the STC8 review). The STC8 manages to execute 82% of its instruction set in a single cycle, and 96% of its instructions in three or fewer cycles. The other parts – the [EFM8](#), older STC15W, and [N76](#) – have much fewer single-cycle instructions. The [EFM8](#) starts to catch up to the STC8, while the [N76](#) remains much further behind.

Multiplies are two-cycle instructions on the [STC8](#) and STC15 – twice as fast as the [EFM8](#) and [N76](#)¹³. The [N76](#) and STC15 have poor performance at bit arithmetic (4 and 3 cycles, respectively) compared to the single-cycle EFM8 and STC8, but this shouldn't affect the results of this experiment.

The big advantage the [STC8](#) has, though, is memory operations. STCmicro gave this part a large, parallel-fetch interface that allows it to read three or more bytes out of flash in a single cycle. That allows it to perform long op-code instructions – like “load immediate value into RAM location” – in a single cycle. This is a three-byte instruction which takes three cycles on the [N76](#) and [EFM8](#). While these architectures are pipelined, they still have a single-byte fetch, which means the length of the instruction is the main factor dictating how many cycles it will take to execute.

When pitted against the 8-bit darling of the RISC movement – the AVR – things get interesting.

On the AVR, there simply is no instruction to do anything with RAM other than move it from/to registers – so to “load immediate value into RAM location” you must first load an immediate value into a register (1 cycle), and then load the register into RAM (2 cycles). This is a three-cycle operation in total.

This may look like there’s no performance difference between the AVR and the 8051, but there’s a major hiccup for the 8051: there are only 128 bytes of true RAM in the 8051 – not enough to hold the 64-word 16-bit data arrays in this experiment.

On the 8051, to access 16-bit extended memory (XRAM), you have to load the RAM address in the DPTR, load the direct value into the accumulator, and then move the accumulator to @DPTR. On the slowest 8051 – the [N76](#) – those operations take 3, 2, and 6 cycles, respectively – 10 cycles total. On the [EFM8](#), those operations take 3, 2, and 3 cycles – 8 cycles total.

Compare that to the two-instruction, three-cycle AVR routine, which can actually store the immediate value to *any* RAM address – all 65,535 of them.

The [STC8](#) comes closest to the AVR: it can perform these XRAM operations in 4 cycles total.

While this is still a penalty to pay when compared to AVR, consider that almost all RAM move instructions – and RAM arithmetic instructions – are single-cycle on the STC8. Since STC8 RAM is no slower than access to the 8 available registers the 8051 has, you get essentially 128 extra registers for free.

While the lousy Keil C51 code prevents you from comparing across AVR and 8051 parts in this experiment, note that the [STC8](#) only needs 153 clock cycles – compared to the 272 and 413 of the [EFM8](#) and [N76](#).

Sure, the [EFM8](#) wins the overall prize in the 8-bit segment in this test. But it chugs to the finish line relying exclusively on its 72 MHz clock, which is running 9 times faster than the [megaAVR](#)’s 8 MHz oscillator, while only performing about twice as fast as it is.

And the [EFM8](#) uses so much power that the standard adage – run as fast as possible to minimize static power – goes out the window. The slower [megaAVR](#) and [tinyAVR](#) both use less power than the EFM8 and STC8.

DMX-512 RECEIVER

POWER	ISR LATENCY	ISR CYCLES	ISR TOTAL	FLASH	FLASH (%)
ATMEL TINYAVR					1430 MICROAMPS
ATMEL MEGAAVR					1270 MICROAMPS
ATMEL SAM D10					3410 MICROAMPS
CYPRESS PSOC 4000S					1030 MICROAMPS
FREESCALE KE04					3340 MICROAMPS
FREESCALE KL03					1340 MICROAMPS
HOLTEK HT66					568 MICROAMPS
INFINEON XMC1100					1390 MICROAMPS
MICROCHIP PIC16					470 MICROAMPS
MICROCHIP PIC24					667 MICROAMPS

MICROCHIP PIC32MM	493 MICROAMPS
NUVOTON N76	1750 MICROAMPS
NUVOTON M051	1970 MICROAMPS
NXP LPC811	3440 MICROAMPS
RENESAS RL78	516 MICROAMPS
SANYO LC87	6420 MICROAMPS
SILICON LABS EFM8	607 MICROAMPS
ST STM8	1620 MICROAMPS
ST STM32F0	769 MICROAMPS
STC STC8	5470 MICROAMPS
TI MSP430FR	279 MICROAMPS

There's a lot going on in this project, so let's break it down separately into performance (in this case, power consumption), development process, peripheral library bloat, and other platform-specific eccentricities.

In terms of performance, the MSP430 is the clear favorite. The next-lowest-consuming part, the PIC16, used almost 70% more power.

Most Arm chips struggled with this test. The SAM D10 was the worst – Atmel START's ASF4 peripheral libraries have an insanely high-level UART abstraction mechanism that took almost 500 clock cycles to process a single byte. Processor Expert's equally high-level interface fared considerably better – executing in 281 clock cycles on the KE04. But both were abysmal compared to the rest of the field.

It was Infineon DAVE-generated XMCLib code that brought in the fastest Arm performance: it took only 72 cycles in the ISR to process data on the XMC1100. The XMC1100 also got closest to the theoretical 15-cycle interrupt latency of the ARM Cortex-M0+. STM32Cube-generated STM32 peripheral libraries also performed well on the STM32F0 – needing only 83 clock cycles to process data.

The Nuvoton M051, with its lightweight, relatively easy-to-use peripheral library did moderately well, but the NXP Kinetis KL03 (running Kinetis SDK) and the LPC811 (running LPCware) both brought in mediocre numbers (102 and 159 cycles, respectively).

As mentioned before, the HT66 has a markedly similar architecture to the PIC16, and yet again in this test, these two 4T parts have very similar performance characteristics.

Halfway through my review, I noticed the PIC16 got really *really* slow. Sure, enough the infamous messages started popping up: “Using Omniscient Code Generation that is available in PRO mode, you could have produced up to 60% smaller and 400% faster code” – my 60-day evaluation license had expired. And they're absolutely right – I had to *double* my device's clock speed to hit the original performance numbers I had. The numbers mentioned in this review were from optimized code, so if you're a student or indie developer (or even a professional who has better things to buy than a compiler), take the results with a *big* grain of salt.

The EFM8 proved to be the most flexible 8-bitter in the round-up. It beat out all 8-bit parts in the high-performance biquad math tests, and it turned around and achieved the third-lowest power consumption figures in the DMX test, too – needing only 607 μ A, while running at a cool 1.531 MHz core speed. Again, a multi-byte FIFO helped immensely.

The tinyAVR and megaAVR both performed poorly – they consistently look like bigger, beefier 16- or 32-bit parts in performance testing: they bring in great math performance and run-mode active current, but they had some of the highest interrupt latencies among 8-bit parts that don't have FIFO UARTs¹⁴. While interrupt latency is a property of the core – not the peripherals – we often measure interrupt latency relative to an actual event happening. Here, peripherals play a major role in determining latency. UARTs with FIFOs in this test had significantly higher interrupt latencies than UARTs without FIFOs., and they both struggle with stop-mode and low-speed run-mode current.

Another 8-bit part that stood out was the STC8, which recorded a 6-cycle interrupt latency – that's 6 cycles from the stop bit being received, all the way to your C interrupt handler executing. Interrupt latency is one major advantage these 8051 parts continue to press over the fancier

RISC-based AVR and Arm cores, which have long prologue and epilogue code. Unfortunately, the STC8 simply does not care about power consumption – so it barrels along at 5.47 mA.

As for ease of development, things were split across the board – even between parts that had code-gen tools, only peripheral libraries, or nothing at all.

With Cypress PSoC Creator, I had to write a custom ISR from scratch, as the generated code didn't contain functions for handling this scenario. I got burned by the function documentation a bit – but was able to get it working after quite a bit of reading.

Infineon DAVE essentially cheated in this competition by providing a pre-built, one-click DMX-512 Receiver Dave APP. Consequently, I simply had to glue its callback to the LEDs by writing three set-duty-cycle calls. It was beautiful.

Excluding DAVE, the easiest to get going was actually the Renesas code generator, which provided function prototypes for me to implement that it called from the interrupt. I have no idea why more vendors don't provide statically-invoked callback functions from their generated interrupt code. They either provide no callbacks at all (STM32CubeMX, Microchip MCC, Cypress PSoC Creator), or they provide a weird, super heavy-handed dynamically-invoked callback register event system (like Atmel), or provide a user callback can't directly handle the (private) data, or manipulate the internal state of the driver – like in the case of Processor Expert on the KE04.

For this particular project I enjoyed Silicon Labs' approach with Simplicity Configurator – it will automatically create the interrupt for you, and will even make sure it gets enabled in start-up code. It will give you helpful comments above the interrupt, telling you which flags to clear. But, the generated interrupt functions are just stubs – they contain no actual code. This is definitely the most flexible and lightest-weight route to take. And while some vendors (Microchip MCC, ST STM32CubeMX, NXP Processor Expert) provide “init-only” drivers, they don't automatically stub the ISR, nor do they automatically enable the interrupt.

The HT66's peripherals were simple to configure by hand – and it required the fewest bytes of flash.

The PSoC was the most efficient of the 32-bit parts – owing to the fact that the majority of code required for this project (peripheral initialization) lives in a compact bitstream representation inside the PSoC.

The Nuvoton runtime libraries are huge (since they're so simple to use), and I'm not positive that CoIDE was properly doing link-time-optimization (I'll need to investigate further).

DISCUSSION

Silicon Labs EFM8: Fantastic value and ease-of-use from the only 8-bit part with a totally-free cross-platform vendor ecosystem

The [EFM8](#) was the fastest 8-bit part in my round-up, and admittedly, my favorite 8-bit architecture to develop with overall. What these parts lack in brains they make up for in brawns – 14-bit ADCs, 12-bit DACs, lots of timers, and a 72 MHz core clock speed that gives you timing options not found in any other part in the round-up.

Plus, this is the only 8-bit part with a totally-free, cross-platform, vendor-provided ecosystem. Let that sink in.

Keil C51 is a silly compiler, but Silicon Labs does an excellent job hiding it under the hood – even when running its Eclipse-based Simplicity Studio on Linux or macOS.

Simplicity Configurator is the lightest-weight code generator in our round-up, using only 534 bytes of flash to house the entire DMX-512 receiver project. It was one of the easiest to use, and seemed to strike a good balance between abstraction, performance, and ease of use.

Debugging speeds are snappy with a J-Link debugger, but at \$35, the official Silicon Labs USB Debug Adapter is one of the cheapest first-party debugger in the round-up, and clones of the hardware are even cheaper.

And call me old-fashioned, but I think the 8051 definitely has a place in 2017 – especially among hobbyists and students, where its bit-addressable memory, easy-to-use peripherals, and fuse-free configuration help get students comfortable with microcontrollers quickly.

Microchip megaAVR & tinyAVR 1-Series: Different strokes for different folks — still with the best 8-bit toolchain available

The [megaAVR](#) came in surprisingly flat for me: especially when compared with its lower-cost, new sibling, the [tinyAVR 1-Series](#).

There's no comparison when it comes to price: tinyAVR has incredible value — packing in a nice assortment of timers, analog peripherals (including a DAC), and a new 20 MHz internal oscillator — while costing 20-40% less than the megaAVR.

While the megaAVR has a perplexing debugging experience that requires two completely different interfaces and protocols to work with the part, the new one-wire UPDI interface the tinyAVR sports worked flawlessly in my testing.

But that's the crux of the problem for the tinyAVR — by shedding many of its megaAVR roots, Microchip ended up with a wonderful microcontroller that will be challenging to use for a large base of Atmel fans: indie developers and hobbyists who use low-cost, open-source programmers (which don't support the UPDI interface).

While the tinyAVR wasn't the fastest part in the round-up (even among 8-biters), it was the most efficient — both in terms of active-mode power and clock efficiency. Amazingly, the AVR only uses about twice as many instructions as 16- and 32-bit parts when performing 16-bit math.

Unfortunately, the AVR system as a whole is not without its issues. The Windows-only Atmel Studio is still buggy (especially with older megaAVR devices and AVR Dragon stuff in my tests), and there isn't an under-\$50 low-cost debugger available (other than hacking apart Xplained Mini dev boards).

In many ways, there seems to be a tacit demarcation Atmel creates between its hobbyist/indie developers, and the professional shops that use Atmel parts.

As a professional embedded developer, I most definitely have access to Windows computers, and I have no problem blowing a few billable hours' worth of pay on a \$140 debugger.

But even as popular as Atmel is among hobbyists, Atmel has largely stayed out of this space directly. Instead, they've secured small-volume AVR sales by relying on the open-source community to build their own tools for themselves: turning out a slew of hardware and software used to program the megaAVR devices.

While I applaud the efforts of these developers, these tools are inferior to Atmel's. Their programming speeds are terrible, they don't support the new tinyAVR 1-Series devices, and they have absolutely no debug capability.

Having said that, both the megaAVR and tinyAVR have the best toolchain available for 8-bit MCU development. The part supports a full, end-to-end Makefile-based GCC toolchain.

If you love `printf()` debugging, would never touch a proprietary toolchain, and hate IDEs, megaAVR and old tinyAVR parts are definitely for you. The older ones are still available in DIP packages, and as you probably know, there are a ton of low-cost programmers available across the world. The online community is massive, and as clunky as I find Atmel START to be, I have to applaud its support for Makefile-based project generation.

Consequently, the megaAVR remains the most open-source 8-bit microcontroller on the market – by a long shot.

But I'd really like to see Microchip provide a PicKit-priced debugger with UPDI support – and allow off-board debugging the way their PIC Curiosity Boards do.

I also hope these open-source projects can add UPDI support to their tools, so that hobbyists and indie developers can start integrating the tinyAVR into their projects – it's a much better part, and if you're an AVR user with access to Atmel Studio, you really ought to buy an Xplained Mini board and take it for a spin.

STM32F0: A low-cost, no-nonsense part with arguably the best Arm development ecosystem tested

The [STM32F0](#) was the lowest-power Arm microcontroller in the round-up, and also one of the easiest to use. STM32CubeMX doesn't generate the most compact code on Arm (that honor belongs to Cypress PSoC Creator and Infineon DAVE), but it has a snappy interface, and the generated code is easy enough to manipulate for your own goals.

I love the nearly-stock Eclipse-based environment that System Workbench for STM32 provides, and the ST-Link and excellent Discovery/Nucleo boards seals the deal for me.

Most pros have used ST parts in their work, but for all these reasons, any hobbyist looking at moving to Arm should probably pick up a dev board from this ecosystem, too. ST has a huge market footprint, so there's tons of resources online – aimed at both hobbyists and

professionals.

SAM D10: Killer performance & peripherals, but with runtime library hiccups

The Microchip/Atmel [SAM D10](#) (and the broader D11/D20/D21 ecosystem) has good value (considering their analog portfolio includes a DAC, and they have good timing options), and the SAM D10 was the most efficient part tested when running at full speed.

Professionals will like the easy-to-use, well-documented header files, and hobbyists will appreciate the 1.27mm-pitch SOIC package options and GCC compilers that come with the Arm ecosystem. But before I grab this part for a project, Microchip really needs to fix the extremely slow, bloated peripheral library, and update their code-gen tool to do proper error-checking of clock and peripheral configurations.

As it is, whenever I use Atmel START on the D10, I want to STOP almost immediately. And there are no current, stand-alone peripheral drivers that Microchip has released for this part, so unless you want to do register programming from scratch, you'll be relying on third-party, open-source projects – like Alex Taradov's [code examples](#).

Infineon XMC1100: Interesting peripheral perks make this Cortex-M0 stand out

The most interesting Arm chip was, without a doubt, the [Infineon XMC1100](#) – and I think professionals who may be wary of getting out of the ST/NXP/Atmel Arm ecosystem need to take a second look at these XMC1000 (and XMC4000) parts.

The timer options are amazingly flexible, and you can squeeze fantastic performance out of the USIC module.

I'm going to go out on a limb and recommend that serious hobbyists who are building motor / lighting control projects look into these parts, too. DAVE makes setting up these complex peripherals painless, and the 38-pin TSSOP chips will be substantially easier to solder than the 0.5mm QFNs and QFPs you usually end up with in these pin counts.

Like many of the parts reviewed here, the biggest problem for hobbyists and indie developers is the tiny online communities and lack of GitHub repos with open-source projects that use these chips. My advice – be bold, and post in the forums. Infineon employees monitor and usually respond within a day or so.

PIC16: Tons of peripherals with a slower, power-efficient core

When you compare the [PIC16](#) with other 8-bit parts out there, it's obviously a part built for low-power applications, and not processing power. And while the development ecosystem is workable, there are other parts more friendlier pathways – especially for smaller shops, hobbyists, and students who need extremely low-cost tools (and free software).

To add fuel to the PIC-vs-AVR debate, my testing found that a 32 MHz PIC16 is roughly equivalent to an AVR part running at 1.4 MHz (in terms of math performance), and 9 MHz (in terms of bit-shuffling performance).

Having said that, the DMX-512 receiver seems a perfect match for the PIC16, and that's where it looks best in my testing: the PIC16 was the lowest-power 8-bit part in my testing.

It's also full of timers and digital logic-oriented peripherals that make it suitable for funky special-purpose projects that require some crafty use of configurable logic and the numerically-controlled oscillator – these peripherals help offload the (relatively slow) CPU, at the expense of requiring more developer familiarity with the device and these peripherals.

The usual Microchip gotchas apply: clunky IDE, expensive compilers, and expensive debuggers.

The usual Microchip advantages apply: huge online community, seemingly infinite product lifetime guarantees, and DIP, SOIC, QFP, and QFN package availability.

PIC24: An expensive MSP430 wannabe that doesn't hit the mark

The [PIC24](#) is nearly forgettable. In the biquad test, it's marginally faster than the [Renesas RL-78](#) but uses almost three times as much power. In the DMX-512 test, both the RL-78 and MSP430 beat it, too. It was also one of the least-endowed parts in the round-up (which really just means it's expensive – higher-end PIC24 parts have no shortage of peripherals).

The usual Microchip gotchas apply: clunky IDE, expensive compilers, and expensive debuggers.

The usual Microchip advantages apply: huge online community, seemingly infinite product lifetime guarantees, and DIP, SOIC, QFP, and QFN package availability.

PIC32: An excellent 32-bit part that balances performance and power consumption

The [PIC32MM](#) was my favorite Microchip part in the review. It brought in the lowest-power performance of every 32-bit part tested. Unfortunately, it was also the least-efficient 32-bit part tested in terms of math performance (well, excluding the couldn't-care-less-about-power [Nuvoton M051](#)), and it's pretty spartan on peripherals – it doesn't even have a hardware I2C controller.

But PIC32MM parts have good flash / RAM density, and have simpler clocking / peripheral gating configurations than some of the more-flexible Arm parts, which makes them feel easier to program at a register level.

Plus, they have a lot of headroom: I think the high-end PIC32MZ DA devices have a home among small industrial dev shops that need Linux-like HMI functionality but don't have the resources to bring a product like that to market.

The usual Microchip gotchas apply: clunky IDE, expensive compilers, and expensive debuggers.

The usual Microchip advantages apply: huge online community, seemingly infinite product lifetime guarantees, and DIP, SOIC, QFP, and QFN package availability.

Renesas RL-78: An agile, low-power, easy-to-use 16-bit part you really ought to try

I had never picked up a Renesas part before, and when I went shopping for dev kits and stumbled on only a smattering of expensive, traditional systems, I was a little anxious. But I found the [\\$25 RL78L1A Promotion Board](#), gave it a shot, and really enjoyed it.

The [RL-78](#) is a snappy architecture that competes with Arm parts in math performance, yet it's also relatively inexpensive – especially compared to the [MSP430](#) and [PIC24](#). It can't quite hit the MSP430 sleep-mode power consumption figures, but it gets close – and is, by far, the most power-efficient 5V-capable part in the review.

The code generator tool produces readable yet efficient code, and the IDE, e2studio, is Eclipse-based – and is getting Linux and macOS support in the next release.

I'd complain about the dev board, but the new [YRPBRL78G13 RL78/G13 development kit](#) should remedy basically all my complains with it – I can't wait for U.S. distributors to start carrying these. They could use a more active community and more people publishing code online, but I hope this article will help inspire some remedies for that.

N76, HT66, and STM8: Low-cost parts with a smattering of development headaches

The [STM8](#) is probably the nicest of the “cheapie” parts. It has nice peripherals and really good performance for an 8-bit part running at its frequency, but I think the entry-level [38-cent STM8S103F2P6](#) is a more compelling part than the higher-end one reviewed here – simply because of its ultra-low price. The part I reviewed here looks a lot like the other 8-bit microcontrollers – but with an ancient-looking IDE that’s not nearly as productive as the competition. And almost everything out there has better power consumption figures.

Still, this part is relatively cheap to get going (ahoy, \$5 ST-Link clones), and the IDE and toolchain are completely free. But in that regard, you get what you pay for: STVD feels trapped in 2002, and there’s no way to set up a more modern development and debugging environment for it¹⁵. Though, there’s some emerging SDCC / GDB support for pairing it with an ST-Link in an open-source fashion, which might help it make inroads with the classic tinyAVR and megaAVR users.

You’ll be forking over quite a bit of money for a Keil C51 license to develop for the [N76](#) – all for a part that doesn’t look much different than some entry-level EFM8s that have a day-and-night difference in ease of development. Still, at [23 cents per unit](#), it’s tough to beat for volume applications – and hobbyists and hackers can probably get by with the 2K code limit of Keil’s evaluation version. SDCC users need not apply: there’s no stand-alone tools for loading code into this part, and I doubt µVision can be coaxed into loading an SDCC-compiled hex file.

The [Holtek HT-66](#) has terrible processing performance, but barely uses any run-mode current – there’s plenty of application-specific models to choose from, and while the IDE is goofy, I found it to be fairly productive – and it’s completely free. Careful, though: only the more-expensive “V” parts have on-chip debugging.

STC8: A neat performance-heavy part for hacking — but probably not for serious, professional work

I think every hacker and advanced hobbyist really ought to throw \$10 at AliExpress/Taobao and get some STC15 and [STC8](#) parts – just for fun.

Both are jam-packed full of peripherals and memory (more than every other part reviewed), and the STC8 is also *really* fast. There are some interesting projects you can do with a part that hits your C interrupt code 10 clock cycles after an interrupt occurs – that’s 320 nanoseconds. Both these parts support debugging over UART, so there’s no proprietary debugger to purchase.

I wouldn’t seriously consider using these parts in U.S.-based commercial work, as we have no access to STC inventory here, but the part is just plain fun to play with.

ON Semiconductor LC-87: Skip

You can probably skip over the [ON Semiconductor LC87](#). This is a rare part outside the Japanese market, and it looks like it's on its way out the door. I called Altium to try to get an evaluation version of the Tasking LC87 toolset, and the person I talked to had never heard of the LC87 before, and was almost positive they hadn't made a compiler for it for at least three years. This part has terrible power consumption, few peripherals, and the worst development environment I saw in this review. Skip.

Kinetis KL03: Sleep-mode specialist not for beginners

While the [Kinetis KL03](#) has excellent deep-sleep current and ultra-tiny CSP package availability, it definitely feels like a specialized part not useful for the applications I evaluated. It has far fewer peripherals than the other parts reviewed, and despite NXP's low-power claims, was consistently in the middle of my Arm rankings for the DMX-512 receiver test – though it nearly matches the [SAM D10](#) in full-speed active mode.

Kinetis SDK is awkward to use, and the dev boards are terrible – requiring a lot of reverse-engineering and hacking to get the board doing anything other than running pre-written demos (especially if you're interested in measuring power consumption). Still, MCUXpresso is a productive, modern Eclipse-based IDE, and the KL03 has some of the lowest-leakage power modes out there, which means you can get 8-bit-like performance when you're running an RTC or interrupt wake-up project from a coin-cell battery.

Kinetis KE04: Decent peripheral assortment with a powerful — yet clunky — code gen tool

The [Kinetis KE04](#) had pretty heavy power consumption in my testing – but this was largely due to the heavy-handed Processor Expert code that Kinetis Design Studio generated. This environment is really suited to much larger, faster microcontrollers running RTOSes and not needing especially good low-level I/O performance.

But, hey, if you don't really care about performance, the nice thing about Processor Expert is it abstracts the peripherals to such a high level that you'll never need to crack open a datasheet for the part if you're using the peripherals in normal configurations.

Plus, the KE04 (and KE02) are 5V-compatible parts, and they're available in old-school, easy-to-solder 1.27mm SOIC and 0.8mm packages – so I could imagine hobbyists would find this part useful.

LPC811: Few perks, and less interesting than the LPC810

The LPC810 drew people in with its odd, 8-pin DIP form-factor. That chip has since been discontinued, but the LPC81x line remains. The [LPC811](#) reviewed here is sparse on peripherals – not even having an ADC – and brought in poor performance. There’s really nothing that this part does that you can’t get from one of the other vendors; but don’t discredit NXP completely – their higher-end offerings have some interesting capabilities (like dual-core Cortex-M4/M0 designs), and their development environment, MCUXpresso, is an inoffensive Eclipse system.

PSoC 4000S & MSP430: Bottom-of-the-barrel parts that offer a glimpse into nice ecosystems

I hesitated to review the [PSoC](#) and [MSP430](#) because they tend to be relatively expensive parts, so in a \$1 shoot-out, you end up with bottom-end parts that don’t look nearly as useful as their higher-cost relatives. If you really want to get a feel for what the MSP430 or PSoC parts can do, I recommend buying into a higher-end part – preferably on one of the excellent dev boards that these manufacturers make.

PSoC Creator and the reconfigurable digital and analog blocks in the PSoC line draw many professional and hobbyist users into the architecture – but instead of grabbing the 4000S from this review, reach for a PSoC5 (or soon-to-launch PSoC6) dev board to get a feel for the platform.

Same with the MSP430. In the DMX-512 test, it dominated in power consumption, but barely put up marks in any other category (this is especially challenging when you have no hardware multiplier, and only a smattering of peripherals).

Still, the part has a solid development ecosystem with Code Composer Studio and a choice between the proprietary (but now free) TI compiler, and the open-source GCC one. Plus, hobbyists will love the easy Arduino migration path (with Code Composer Studio directly supporting Energia *.ino sketch projects) and \$10 dev boards.

And really, everyone starting a battery-based product needs to go buy an MSP430 Launch Pad and play around with it – these really are amazing parts that still have a lot of relevance in 2017.

Nuvoton M051: Ecosystem issues stifle a performance-packed part

The [Nuvoton M051](#) – one of the most-endowed parts reviewed – suffers ecosystem issues that Nuvoton could easily remedy in the future, so I’ll reserve judgment. There’s no manufacturer-provided Eclipse-based IDE – instead, the only IDE options are Coocox, and Keil μ Vision – neither of which I’m particularly fond of.

I was able to get Coocox working (though the peripheral libraries that are in the Coocox repo are old and full of bugs). The M0 had some of the worst power-consumption figures in the review, but it makes up for that with tons of communications peripherals, beautiful 32-bit control-friendly timers, and easily-digestible runtime libraries and documentation that are far easier to use than other vendors’. When Nuvoton fixes the IDE absence, I’ll definitely move this part from the “meh” to “yeah” column – since it accomplishes all of these feats while remaining one of the lowest-cost Arm microcontrollers out there.

CONCLUSION

I had a ton of fun playing with all these different parts for the last few months for this microcontroller review, and in many ways, came away thinking what I already knew: there is no perfect microcontroller – no magic bullet that will please all users. What I *did* learn, however, is it’s getting easier and easier to pick up a new architecture you’ve never used before, and there have never been more exciting ecosystems to choose from.

And that’s what I want people to think about as they walk away from this microcontroller review. If you’re an Arduino hobbyist looking where to go next, I hope you realize there are a ton of great, easy-to-use choices. And for professional developers and hardcore hackers, perhaps there’s an odd-ball architecture you’ve noticed before, but never quite felt like plunging into – now’s the time.

It's an exciting time to be involved with electronics – whatever parts you choose to pick up, I hope you've enjoyed learning about what's out there, and can get inspired to go build something great. Definitely leave a note in the comments below if you've got something to contribute to the discussion!

Footnotes

- ↑ To get technical: I purchased several different MCUs – all less than a \$1 – from a wide variety of brands and distributors. I'm sure people will chime in and either claim that a part is more than a dollar, or that I should have used another part which can be had for less than a dollar. I used a price-break of 100 units when determining pricing, and I looked at typical, general suppliers I personally use when shopping for parts – I avoided eBay/AliExpress/Taobao unless they were the only source for the parts, which is common for devices most popular in China and Taiwan.
- ↑ There's considerable debate as to the precise definition of a "RISC architecture" is, but while the PIC16 has a single-word instruction length for all instructions, the PIC16 varies greatly from most RISC parts in that it is an accumulator-based machine, and has no working registers. I'll leave it up to you to decide.
- ↑ Formerly ARM, but as of August 1, 2017, "Arm" is the capitalization style [they now use](#).
- ↑ Advanced Microcontroller Bus Architecture – these multi-level acronyms are getting tedious
- ↑ the 8051 is a member of a family officially called the "MCS-51" – along with the 8031, 8032, 8051, and 8052 – plus all the subsequent versions that were introduced later
- ↑ Expensive windows ceramic packages allowed [EPROM programming](#) for developers, but production units were mask-ROM or OTP – or, in the case of the 8031, only external ROM was supported.
- ↑ The only other mainstream MCU that has an integrated Ethernet PHY is the \$14 Tiva-C TM4C129x, a giant 128-pin 120 MHz Arm Cortex-M4 from Texas Instruments. There are a few other (albeit odd) choices out there: Freescale's legacy ColdFire microcontrollers include the MCF5223X, which has an integrated Ethernet PHY. Fabless designer ASIX manufacturers the AX11015, a 100 MHz 8051 with an integrated Ethernet PHY
- ↑ and a [growing chorus](#) is [questioning](#) whether they're in violation of GPL

9. ↑ Full disclosure: Raisonance has a stake in GCC, as they use it in their Ride7 proprietary IDE.
10. ↑ PL/M-51 wasn't a C compiler – it actually compiled code written PL/M, a proprietary Intel high-level language. Man, the 80s were weird.
11. ↑ Yes, I know you can disable these extensions with the NOEXTEND compiler directive... but obviously then you can't use these directives.
12. ↑ xdata
13. ↑ The [N76](#) and STC15 have poor performance at bit arithmetic (4 and 3 cycles, respectively) compared to the single-cycle EFM8 and STC8, but this shouldn't affect the results of this experiment.
14. ↑ While interrupt latency is a property of the core – not the peripherals – we often measure interrupt latency relative to an actual event happening. Here, peripherals play a major role in determining latency. UARTs with FIFOs in this test had significantly higher interrupt latencies than UARTs without FIFOs.
15. ↑ Though, there's some emerging SDCC / GDB support for pairing it with an ST-Link in an open-source fashion, which might help it make inroads with the classic tinyAVR and megaAVR users



COMMENTS (59)



CHARLES

November 6, 2017 at 1:18 pm

REPLY

Purchase links would be appreciated 😊



ALAIN

November 6, 2017 at 1:19 pm

REPLY

Impressive work! I'm a big user of Atmega, SAMD and STM32F0, and I learned a lot in this article. I agree that SAMD is a great series of chips that comes with poor libraries. STM32F0 has better libraries but peripherals are sometimes a bit quirky... So it's good to play with both worlds. And I see from this article that there is a lot more!



STEF

November 6, 2017 at 1:21 pm

REPLY

well done, great comparison, appreciate Your work, Thanks a lot.



PETER

November 6, 2017 at 1:38 pm

REPLY

A wonderful article! It would have been interesting to include the EFM32ZG (<https://www.silabs.com/products/mcu/32-bit/efm32-zero-gecko>) in this comparison. Another ARM Cortex-M0+ that can be had for just under a dollar with 4KB flash and 2KB RAM.



PETE

November 6, 2017 at 2:30 pm

[REPLY](#)

Detail-packed review! I'd often thought I'd like to do a series on various low-cost MCUs and their ease of climbing the complexity ladder from blinking a LED to something more advanced. Your review here covers nearly everything I would have done and then some. Thanks!



ZACH FREDIN

November 6, 2017 at 3:13 pm

[REPLY](#)

Excellent article. I really appreciate the attention to detail – peripherals, firmware, packaging, toolchain, everything.

I've had good luck with the relatively new STM32L0 series. Might be worth a look – it addresses the above-average power consumption concern you listed for the F0. I have used the cheaper versions (STM32L011x) in a few designs and it's a nice little chip. And cheap.

DAVID L JENKINS



November 6, 2017 at 4:50 pm

REPLY

Great work and very much appreciated. I will take a few days to digest it. But I still can't believe that you can get a powerful machine like this for only a dollar.



DANIEL

November 6, 2017 at 7:47 pm

REPLY

Very interesting and detailed post. It pretty much agrees with a lot I've experienced myself. Just two cents from my side: It's not just Atmel Start which is horrible but the whole peripheral section of the ATSAMD are rather shitty with their dozens of unsynchronized running clocks which tend to lock up the system rather easily and hard if one forgets even just one of the obligatory waiting loops. Just thinking of them makes me want to cry out loud in horror, only emphasised by the largely incomplete documentation with thousands of errors and discrepancies and a complete lack of examples both in and outside of documentation. I will gladly take any opportunity I get to stay the hell away from them.

The other thing I wanted to add is that STM32CUBEMX can not only generate code for the rather sizeable HAL API (it's actually a nice touch that it can also generate Makefile projects and include all required APIs for that), but also code for the much slimmer (and hence for me) more useful LL API.



KAT

November 6, 2017 at 7:56 pm

REPLY

What about ESP8266 ?



BOB SEELEY

November 6, 2017 at 9:48 pm

[REPLY](#)

Incredibly well done. It certainly has opened up a lot of information that I didn't know existed. Thank you very much.



BYRON SHILLY

November 6, 2017 at 10:47 pm

[REPLY](#)

Quality piece. Keep up the good work.



PATRICK O

November 7, 2017 at 8:24 am

[REPLY](#)

This. As popular as the ESP8266 is, I'm also disappointed it wasn't included here. Probably the lack of documentation. Nonetheless, this was a great write up.



PAUL GEORGE

November 6, 2017 at 10:50 pm

REPLY

👊👊👊 Bro.

Came here as a part of the hackernews hug of death . Technical Literature of this kind is badly needed by the embedded community. Hats off , In agreement with most of the discussion on the hacker news post . I struggled with the selection of a microcontroller for a product I had to develop, and SEO in this domain is poor .

You've Earned yourself a Email Subscriber.

I'm personally engaged with the development of RISC-V Cores as a part of the Shakti Processor project at IIT Madras . I'd love to lend a hand towards the extensive homework needed to keep these going 😊 !!

Kudos !



AUTARCHEX

November 6, 2017 at 11:49 pm

REPLY

This is an amazing and thorough treatment, thank you! This beats the pants off every MCU roundup I've ever seen.

I would like to suggest a minor correction to the ARM Cortex-M0 description, though:

“Another problem with ARM processors is the severe 12-cycle interrupt latency. When coupled with the large number of registers that are saved and restored in the prologue and epilogue of the ISR handlers, these cycles start to add up.”

As I understand it the majority of those 12 cycles *are* the save/restore of registers on entry/exit of the ISR, which is performed automatically by hardware. The use of ‘when coupled with’ implies that the 12 cycle hit is extended beyond this, but unless the compiler is incredibly naive or the ISR is (too) complicated, there should be no prologue/epilogue code and no additional latency.



STEPH_TSF

November 7, 2017 at 9:41 am

[REPLY](#)

autarchex – Sorry buddy, you understood it the wrong way. The interrupt latency takes 12 cycles, for your interrupt code to start executing. And, speaking of a microcontroller that is not featuring a shadow register set, your interrupt code is supposed to start with the IRS prologue (possibly automatically generated by the compiler) that’s saving the context (all CPU registers that your ISR does modify).



DANIEL

November 9, 2017 at 8:03 am

[REPLY](#)

Err, no:

“To make the Cortex-M devices easy to use and program, and to support the automatic handling of nested exceptions or interrupts, the interrupt response sequence includes a number of stack push operations. This enables all of the

interrupt handlers to be written as normal C subroutines, and enables the ISR to start real work immediately without the need to spend time on saving current context.”

[<https://community.arm.com/processors/b/blog/posts/beginner-guide-on-interrupt-latency-and-interrupt-latency-of-the-arm-cortex-m-processors>]

However it's not 12 cycles but 15/16 minimum for Cortex-M0.



RICH WEBB

November 7, 2017 at 2:52 am

REPLY

Yes, excellent write-up. It's going to take more than one read-through to digest it all.

Bummer that the Arm M0 chips don't include bit banding, which would have covered the bit-addressable register topic, but one can't have everything.



E SHULL

November 7, 2017 at 8:41 am

REPLY

Incredible writeup; thanks for putting this out there!

Because sometimes I can't avoid proofreading, a couple formatting issues I noticed:

* Under the Peripherals comparison section, the notes for the PWM tab are on the previous (timers) tab.

* In the discussion of code generation for the ST STM32CubeMX, you appear to have an inadvertent paragraph break in the middle of the sentence “The other problem is how basic some of the periphe”



JAY

November 7, 2017 at 4:41 pm

REPLY

Actually, the TimerMark score includes PWM channels as part of the criteria; the separate PWM tab is just for convenience – I will add some clarity to that section in the future. And thanks for spotting that random line break! Appreciate the editing help 😊



STEPH_TSF

November 7, 2017 at 9:45 am

REPLY

Do all 8051 clones (reviewed here) feature shadow registers sets (register bank switching)?



JAY

November 7, 2017 at 5:12 pm

REPLY

Thanks for the good question. While the EFM8LB1 and N76 do, the STC8 actually maps some of its peripherals into XRAM (controversial, but considering most of these register interactions are one-time initialization, I'm not opposed to the slower speed).

Lower-end EFM8 parts – like the Busy Bee (EFM8BB1) – do not, either (since all their peripherals fit into the register set).



GARY BERGSTROM

November 7, 2017 at 10:23 am

REPLY

Nice article.

Of course I've got a comment about the micro I use most – the MSP430.

You can get a quite full featured (a/d, dma, multiplier etc) part in the family for \$1.10.

Not quite the \$1 price of your eval. And if you let that number creep then you get thousands of parts. And it gets more difficult to do any kind of eval.

I'm moving to the Arm Cortex parts and I'm pretty happy with the newer Cypress devices. But most of the things I look at cost much more than \$1. I do lower volume, harder to do things. The cost of the CPU is way down the list. IDE tools may be the most important as development costs dwarf the micro cost when you only build 1000 (or less!) of something.

But again, nice article.



JON SMIRL

November 7, 2017 at 1:53 pm

REPLY

What about the ESP32? About \$1.50

Another group, the ONVIF camera chips.

HI3518e, GM8135S – ARM9 + 128MB RAM, \$3.50

Allwinner V3S – Cortex-A9 + 128MB, \$3.50

These chips contain two dies inside the package.



JAY

November 7, 2017 at 5:36 pm

[REPLY](#)

These are all interesting parts that deserve study (especially those Allwinner chips that integrate DDR with the CPU), but they're all way more than \$1, and really have little to do with anything else on the page. Thanks for the recommendations, though – I'll have to do a future project with one of these devices.



JOSEPH CHIU

November 7, 2017 at 2:42 pm

[REPLY](#)

Wow, amazing body of work! I fully agree with your PSoC assessment – if you can't make use of the virtual peripheral and touch sense capabilities, it's far less interesting.

The value of a good toolchain and IDE cannot be stressed enough for people trying to “grow up” past using the standard Arduino environment. You get so much more accomplished with a proper environment to write and debug code in!



DOCTOR WIZARD

November 7, 2017 at 3:22 pm

[REPLY](#)

If you are not already familiar with it, you should check the VisualMicro extension for Visual Studio. It works with any controller that has an Arduino-like GCC/Wiring platform. And you get intellisense and all the other bells and whistles of Visual Studio. (Great if you are already used to Visual Studio and do non-microcontroller related stuff too). I use it with AVR, Arm, STM8 and STM32, ESP8266 and ESP32, ChipKit and PIC32.



MATHEUS

November 7, 2017 at 4:41 pm

[REPLY](#)

Congrats for the really amazing work on this comparison!

There is so much information here that I haven't had enough time to read everything yet but sure I will

I'd like to know which tools did you use to get the current measurements. I'm in need of a tool to provide this kind of information while developint at my work.

Regards,



JAY

November 8, 2017 at 1:36 am

REPLY

Thanks! I used a Silicon Labs STK for current measurement. I've updated the article with this information.



BOZ

November 7, 2017 at 7:33 pm

REPLY

My only real bug-bears on the PIC MPLABX IDE is the removal of the optimization on the non-paid for C Compilers which happens to the 32 bit and 24 bit compilers as well as the 8 bit ones!



JAVIER CANO

November 8, 2017 at 2:38 pm

REPLY

Congratulations for this EPIC uCs review Jay, kudos for you!

I use to dig with ATmegaAVRs just with command line tools (avrdude and the good old Nano editor) on my Mac, so IDE stuff is no issue at all for people like me, and believe me, there is a lot more fun in a plain terminal than in IDEs.

You have one more email subscriber!

Best regards.



MOHAMED FEZARI

November 9, 2017 at 2:42 am

[REPLY](#)

Impressive work, great comparison , we apreciat this work that can help in choice of microcontroller to design an application. I have learned a lot. Many Thanks



BAH

November 11, 2017 at 10:13 am

[REPLY](#)

This is one excreably crappy site. There's text but I can't read it because all the shitty reinventing-the-obvious with much too much javascript.



TROLL DETECTOR

January 9, 2018 at 10:34 am

REPLY

Troll Alert



CAMERON

November 13, 2017 at 3:44 pm

REPLY

Cheers Jay,

Thank you for such a deep comparison. Judging from the other things on your site, I'd love to see a write-up from you on embedded Linux / RTOS functionality on the bigger brothers of some of these. You've got me as a subscriber, no matter what you're working non next.

-cb



GEORGE A. CHAPMAN

November 14, 2017 at 10:20 pm

REPLY

AVR

1. "And interrupts are one of the weak points of the AVR core: there's only one interrupt priority, ..."
tinyAVR 1-series has two interrupt priority levels (datasheet, CPUINT)

2. like megaAVR, tinyAVR 1-series is Harvard architecture but with unified memory (datasheet, Memories, Memory Map)

3. tinyAVR 1-series UPDI is more capable than megaAVR debugWIRE

Parametric Reach

megaAVR

XMEGA AVR are a follow-on to some megaAVR.

—

ATmega2561 – 16MHz, 64 pins, 256KB, 8KB

ATxmega384C3 – 32MHz, 64 pins, 384KB, 32KB

—

ATmega1280 – 16MHz, 100 pins, 128KB, 8KB + 56KB on EBI

ATxmega128A1U – 32MHz, 100 pins, 128KB, 8KB + 16MB on EBI

Atmel Studio

“The excellent IntelliSense engine that Microsoft spent years perfecting has been replaced by some sort of Atmel-proprietary

“Visual Assist” technology that ...”

1. IntelliSense and AVR is available from third party(ies)

2. Visual Assist from Whole Tomato Software

Microchip Customizations

“As these compilers [XC] are quite expensive and ...”

About 1USD/day though the dongle is 1495USD

Discussion

megaAVR

“While the megaAVR has a perplexing debugging experience that requires two completely different interfaces and protocols to work with the part, ...”

fyi, once debugWIRE is enabled then stay with it until done.

“... low-cost, open-source programmers (which don't support the UPDI interface).”

pyupdi is Python UPDI programming via an inexpensive USB UART; debug via UPDI is currently only Atmel-ICE or Power Debugger

PIC32

“... expensive compilers ...”

Can chipKIT PIC32 GCC be invoked from MPLAB X?

Thank you!

Notes

1. IntelliSense with most AVR is available from VisualGDB
2. IntelliSense is in Visual Studio Code and AVR might be available via the Native Debug extension
3. The next release of PlatformIO may restore AVR debugging; Visual Studio(s) are some of the IDE that integrate PlatformIO
4. LLVM AVR may be a current or future alternate to AVR GCC
5. ATmega328PB is approaching 1USD each for 100 though it has a supply and demand problem that's easing 17Q4 (should be resolved 18Q1)
6. 32KB tinyAVR 1-series is likely for '18 and probably under the price limit

Ref.

VisualGDB, <https://visualgdb.com/tutorials/avr/>

Visual Studio Code, <https://code.visualstudio.com/>

Native Debug, <https://marketplace.visualstudio.com/items?itemName=webfreak.debug>

PlatformIO, AVR, <http://platformio.org/platforms/atmelavr>

PlatformIO for Visual Studio Code, <https://marketplace.visualstudio.com/items?itemName=formulahendry.platformio>

Whole Tomato Software, <https://www.wholetomato.com/default.asp>

XC8 PRO subscription, <http://www.microchip.com/Developmenttools/ProductDetails.aspx?PartNO=SW006021-SUB>

Featured MPLAB XC Compiler Licenses, http://new.microchipdirect.com/product/Cat10_SubCat12

LLVM, Changes to the AVR Target, <http://releases.llvm.org/5.0.0/docs/ReleaseNotes.html#changes-to-the-avr-target>

Python UPDI driver for programming "new" tinyAVR devices, <https://github.com/mraadvark/pyupdi/>

chipKIT, <http://chipkit.net/>



November 19, 2017 at 12:42 pm

REPLY

Some little nitpicks:

> From left, the STM32, PSoC 4000S, Microchip SAM D10, Silicon Labs EFM8, and NXP Kinetis KL03 development boards.

The first board on this photo is actually an STM8 board. You also tell that STM32Discovery is a snap-apart board – that’s again an attribute of STM8Discovery.

> I also hope these open-source projects can add UPDI support to their tools

There is an open-source UPI programmer already! And it is really simple to make: <https://github.com/mraardvark/pyupdi>

Unfortunately, I don’t have new tiny parts lying around, so I couldn’t test this one.



BRETT SMITH

November 19, 2017 at 1:07 pm

REPLY

Great write up, confirms my feeling that the STM32 is the right choice for my next project. One small corrections.

One small correction. DMX-512 has 512 bytes after the start bit (the “0”) not 511. The “0” is an optional start code to say that the data is something other than raw control data.



PAUL CARPENTER

November 21, 2017 at 10:31 am

REPLY

Actually yes DMX-512 is up to 512 bytes after a START code like a '0' See standard available from <http://www.plasa.org>

However the '0' is NOT optional it is mandatory as this is the Dimmer ClassData stating that up to the next 512 bytes are 8 bit fader levels.

The START code MUST be processed and actioned appropriately depending on its value and should NEVER be assumed to always be '0' (NULL as defined in the standard). It is never optional.

DMX-512 Annex D states other start codes are reserved with codes in DECIMAL

23 ASCII text packet

85 Test packet (network test)

144 UTF-8 Text packet

146 – 169 reserved

171 – 205 reserved

207 System Information Packet

240 – 247 experimental do not use



MORSY

November 19, 2017 at 2:23 pm

REPLY

Thank you



RANDO

November 20, 2017 at 7:54 pm

REPLY

Double Thank You!



ANDY

November 20, 2017 at 8:48 pm

REPLY

Great post, very interesting. I'm a big fan of the MSP430 myself, mainly for the low power capability. I find them relative easy to program/debug with GCC.

Though its not a \$1 part TI have a special offer(\$4.30) on the new FR2433 Launchpad if you want to grab some, I just got 2. I have just built a custom board with mighty MSP430FR5994., 4 UART and 4SPI/I2C. Nowhere near 1\$ but makes me smile.

Thanks again



TOMASZ

November 21, 2017 at 9:44 am

REPLY

Very good job! And I'd have suggestion to make another review of 'under fiver' wireless SoC's. Whatever usable for sensor apps etc. Recently I started using Nordic nRF52 and I'm impressed by these Arm-CortexM4 capabilities, say nothing about low power power

operation and excellent wireless (I use Nordic custom protocols and shy away from BLE stack) . All important is excellent SDK with examples, all producing small efficient code. Cheers!



JOHNNY HANSEN

November 21, 2017 at 2:14 pm

REPLY

Really impressive work.

Please consider Renesas Rx100 series as well.
Extremely core efficient and <1USD as well.



MARTIN KLINGENSMITH

November 29, 2017 at 6:47 am

REPLY

The PIC16 discussion is nostalgic to me because I spent several years writing assembly code for PIC16 and PIC18. The old version of MPLAB before it became NetBeans was very quick. Low on fancy features, but you could compile/program/debug in seconds. The newer MPLABX was a tough pill to swallow.

Also I'm glad to see you reinforce my experience with Atmel START. Should be called Atmel STOP.



LARRY AFFELT

December 1, 2017 at 8:45 pm

REPLY

Great article with good comparative analysis! I started out with the Intel '51 cores (still have my UV eraser), thru the Phillips/NXP '51 OTP and flash chips, and now my company is a Microchip house. Yes, the debugging has significant delays but I have learned to live with that, I've had worse! I have used everything from 8 pin PIC12F's to 144 pin PIC32MZ parts and found that the upgrade paths are rather easy as the pinouts rarely change within the same package, and once you get used to the IDE and have a good code base to work from I can develop new designs fairly quickly. Besides, once you get used to an environment it is such a pain to switch to another manufacturer's tools ...

Thanks, I am looking forward to reading more of your website and updates!



BERWYN

December 1, 2017 at 11:17 pm

REPLY

This is impressive. I'm a complete fan.

One thing missing is packaging – from the comparison tables. Many projects need to make choices based on this. Depending on project, selection commonly includes conflicting criteria:

- smallest package
- most I/O pins
- hand-solderable

Having these in a comparison table would be very useful.



AJOY RAMAN

December 2, 2017 at 6:06 am

REPLY

Wonderful work!

I have been using PIC devices which are 8-10 years old and experimenting with the Arduino and MSP430 series. Your excellent comparative study will be a great help while migrating to current devices.

– IDE licencing aspects could be a point worth enumerating.



MICHAEL

December 2, 2017 at 9:33 am

REPLY

Hi

I'm quite impressed by the work you have done. It's a lot of material to read and reading it reminded me how many hours i have spent in frustration.

As for my personal point of view what is the best to choose . Well i believe every experienced engineer finds it's way to work effectively with mcu weaknesses after crossing the 8 bit gap and moving to 32 bits .As to what is the best to use the most important thing is "Second source " or compatible chip. Strange you might think but bottom line that's the weak point of all MCU vendors

No matter how good or bad a chip is the menace of all is when you have to buy at double price or wait 4-6 months . Using MCU's as hobby is ok but when you decide to make a leaving things change . Based on that i believe the choise is clear among various chips you examined .

Thank you for the good job you made !!!!!



ALVIN P SCHMITT

December 3, 2017 at 4:11 am

[REPLY](#)

Do you have a .pdf of this paper?



FREESCALEISDEAD

December 6, 2017 at 3:19 pm

[REPLY](#)

Great work. And I applaud you for having the guts to state the sad truth about the direction NXP has chosen with their Espresso toolset. Abandoning Processor Expert is a crime. It has been one of the best tools of its kind. I understand their need to have a toolset that supports all of the legacy NXP processors, but they are penalizing all of the Freescale users in the process. It's going to make me think twice on what platform I use for future projects. ST is looking quite good at this point!



ANDREW KOWALCZYK

December 17, 2017 at 10:06 pm

[REPLY](#)

I just want to say thank you, I've been reading over this for the past two days and I feel like I could keep doing that for weeks or months with all the excellent information here. I've worked a lot with msp430s and STM32s lately, but I now really want to try the SiLabs 8-bit and 32-bit offerings (even though they're not listed).

One of the things that I think this addresses better than any other "getting started" is the toolchains and debuggers, I'm a huge fan of Eclipse-based offerings like Code Composer and Atollic TrueStudio, so anything with a similar debug setup seems like a good idea to try out. Same goes for debug tools: this is the first time I've seen debug tools as a consideration for a comparison for part usability, and it echoes the same problems I encountered trying to work with the AVR and PIC parts on Windows.

Thanks again, and I look forward to seeing future projects.



BRANDON

December 27, 2017 at 9:19 am

REPLY

Great comparison. I'm an Atmel fan myself – Atmel Studio has been nicely stable for me for the last couple years, although I remember when it was far worse. And the AtmelICE is actually a deal for a hobbyist – for \$140 (less if you skip some accessories), you have complete source-level debugging to all AVR and SAM devices – that's a big tool to put in the toolbox.

Atmel ASF can be good or bad. I like the minimalist approach on AVRs – sure, it doesn't give you structure bit fields, but I know what to do to get that kind of operation anyway. On the other hand, some of the SAM ASF packages have been nightmares to work with – confusing, bloated and hazy documentation (I'm looking at you, Cortex M0+ parts...).

I was glad to see information about Renesas – that's one brand I've wondered about but thought it was kind of out of reach. Maybe it's worth a closer look now.



ASDF

January 25, 2018 at 4:42 pm

REPLY

Nitpick: The PIC32MM uses the microAptiv UC core, not M4K like the older PIC32MX.



JIM K.

January 29, 2018 at 1:09 pm

REPLY

Great article...lots of good work!!!!

The MSP430FR2111 is out of the price range for this shootout.

I looked on Findchips.

Arrow has it for \$1.38.

Digikey sells quantity of one at \$1.25.

Your chart lists the Flash size at 16K.

It's actually a poultry 3.75 K.

A more comparable part is MSP430FR2433

15.K of Flash and 4K of ram...\$2.27 for quantity 1 from Arrow.

Not a cheap family.

The new ATTiny chips run from \$0.75-\$0.98 for quantities of one at Digikey.

I'm waiting for the 32K part to hit the market.



DAVID

January 30, 2018 at 8:55 pm

REPLY

Excellent!

But I have a double about the "PERIPHERALS/FLASH/Holtek HT66 8K"

Which MCU do you select? The HT66F0185 have only 4K flash. And the new HT66F019 have 8K.

Thanks.



DEAN

February 2, 2018 at 9:55 pm

REPLY

Excellent write up, thank you for investing the time, extremely useful. Would be great to see the comparison with ESP32, or another article on the wireless chips.



JONATHAN

February 20, 2018 at 9:56 am

REPLY

Hi, thanks very much for your excellent write-up. Just a point to note – since ST bought Atollic at the end of 2017, the TrueStudio IDE is now free for STM32s. Paired with STs ST-Link V2 debugger, it makes a nice (and cheap) development environment for the STMs.



EELCO

February 24, 2018 at 3:40 am

REPLY

Dear Jay,

I can't thank you enough. Bringing together all of this information is absolutely great! Yet, like everyone else I have some personal preferences when choosing an microcontroller. I'm an old fart and grew up programming the MC6809 CPU in assembly with it's lovely CISC core. Fast forward to 2018 if I had to choose now I would probably not go for a ARM cortex-m0 MCU but look toward the S12Z MagniV MCU series from NXP (Freescale). Unlike the older S12X series from NXP the S12Z has a nice CISC CPU core with real 32-bit registers and a real 24-bit programcounter so swapping pages is not needed any more. These are automotive parts and can withstand some electrical abuse. The errata from the S12ZVL (\$2,45 @ DigiKey) is only six pages long! I know in this day an age one should be programming in a high level language like C and not pay attention to what is underneath. But I like to know what's underneath and still be able to program the things underneath in assembly language.



HOBBY16

February 25, 2018 at 1:46 pm

REPLY

Thank you for your awesome review, simply awewome!

About the STM8S that I know well, I must add that what it makes it very attractive is there are plenty of cheap Chinese functioning modules based on it, thermostat with led display, adjustable power supplies, pwm generator... You can buy them for peanuts and then reprogram them for your own purpose. For example, a STM8S003F3 based thermostat board with led display, buttons, relay and NTC costs less than 2\$ (!). See a (very partial) list here : <https://github.com/TG9541/stm8ef/wiki/STM8S-Value-Line-Gadgets>

Another thing, you said “in that regard, you get what you pay for: STVD feels trapped in 2002, and there’s no way to set up a more modern development and debugging environment for it”. That would be unfair if you don’t mention the IAR IDE, which let you do codes up to 8kB which is precisely enough for the STM8S003F3 & STM8S103F3 which has 8kB. It’s works at the get go, you can use it to compile a very efficient code and debug in realtime with the ST-Link dongle’s SWIM interface. I use the IAR IDE to develop and reprogram a digital Volt-ammeter to make the “Charge Doctor”, a product quite well known by electric wheelers :

<http://hobby16.neowp.fr/2016/11/07/charge-doctor-v2-firmware-v-2-03-2/>

All the code is less than 8kB, something I could not cram in using the SDCC, so IAR has a really really good IDE and compiler, all for free, as long as the code doesn’t exceed 8kB.

Last word, the STM8S003F3 has been so successful that Nuvoton has now its pin-to-pin compatible SSOP-20 chip N76E003. It seems a good replacement, with 12 bit adc (instead of 10 bit) and 16k (instead of 8k) and costing just around 15 cts, half the price of the STM8S003F3 (!).

The STM8S003F3 in some of the modules above are now shipped with the N76E003. I have bought a batch of one hundred Volt-ammeters expecting them to have the STM8S but instead they have the N76E003, it was quite a bad surprise. Now it would be neat if I can find a way to port (and debug, if possible) my code to the N76E003 without paying a full fledged IDE from Keil or IAR. I’m still searching, if someone knows...



LEAVE A COMMENT

Message *

Name *



Mail *



Website



SEND COMMENT

Notify me of new posts by email.

Navigation

RULES

CONTENDERS

SPECS



Core

Peripherals

Parametrics

ECOSYSTEM



Dev Environments

Code Generators

Compilers

Header Files

Dev Tools

PERFORMANCE



Bit Toggling

Biquad

DMX Receiver

DISCUSSION

CONCLUSION



© 2017 Jay Carlson