Applying ATS to Embedded Microcontrollers

William Blair, Hongwei Xi

Motivation and Implementation

Microcontrollers interact with the physical world, have few resources, and have about enough abstraction for C. ATS shares C’s native data representation but features a rich type system to verify constraints in a programmer-centric way.

We evaluate ATS on the 8-bit ATmega328p chip found in the popular Arduino platform. We argue ATS can be used for abstracting hardware and static verification.

Example Applications

- Interrupt Driven I/O (USART/TWI)
- Timers
- Pulse Width Modulation
- Analog to Digital Converters
- Interrupt Driven I/O (USART/TWI)

Manipulating the Interrupts Enabled Flag

Atomicity on AVR

1. Clear the global interrupt enabled flag
2. Go through the critical section
3. Remember to restore the interrupt enabled flag

If we forget, we risk a deadlock. Abstract linear types help prevent this. No-runtime overhead, simply proof terms.

Utilizing Hardware Timers with Templates

For reactive devices, periodic tasks are a necessity and hardware timers are all we have. In normal mode, a timer increments in step with a scaled clock and overflows at its max value.

If an interrupt occurs at every overflow, we can run a task every

\[ \frac{N \times 256}{f_{\text{clk}}} \]  

Where N is the prescale factor (1,8,64,256, or 1024)

Support Arbitrary Periods

1. Calculate the number of ticks in desired period, set as threshold
2. Setup a counter in software
3. Increment at each overflow
4. When a threshold is exceeded, execute the task, reset counter

Configuration is unlikely to be uniform across MCUs, but the algorithm stays the same. We parameterize the schedule function on the desired unit of time, MCU, and timer. This allows us to use one copy for the algorithm and defer deciding a threshold to a template function. If the user wants full control, they can implement it themselves and ATS will choose theirs over a default during template instantiation. This example runs a task every 100 ms.

\[
\begin{aligned}
\text{amuxtytype } & \text{transaction} ( & \text{sum:int, messages: int} ) \\
\text{fun add_msg} ( & \text{sum, n, sz:nat | v >= 2;} \\
& \text{valid_transaction(sz|sum, n)} ) \\
& \text{trans:itransmission(sz|sum, n)} \\
& \text{size:int sz,} \\
& \text{trans:transaction(sum|n) } \\
& \text{transaction(sz|sum, n+1)} \\
\end{aligned}
\]

Constraints:

- A transaction partitions a buffer into n messages
- Every message must have at least two bytes
- The format cannot exceed the driver’s buffer
- Upon start, interrupts are enabled and the device has been initialized. Thereafter, the device is busy

Comparison to Plain C/C++

- Thanks to GCC optimizations, the timer example’s final binary size compares with a version written in C
- Encouraging, but richer comparisons are needed between both C and C++
- g++ may provide better in-lining since C++ uses templates as well.