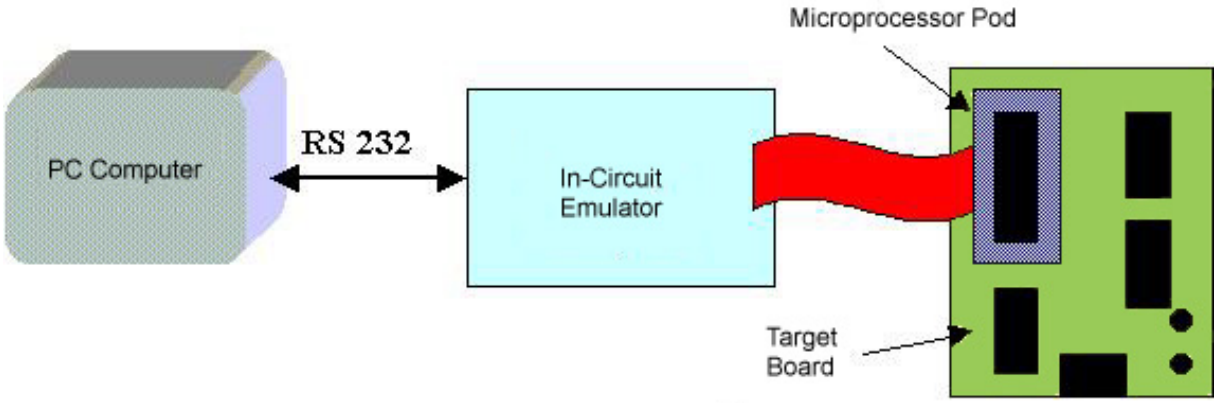
Microprocessor Systems Laboratory

In-circuit Emulators

What is an In-Circuit Emulator ?

An in circuit emulator, often called an ICE, is an invaluable software developers tool in embedded design. The processor or microcontroller of the target hardware will be replaced by the ICE. Often a smaller part of the emulator, the pod, is put into the hardware, while the main emulator functionality resides in a box which is connected to the pod with cables.
An ICE can emulate the replaced processor or uC in real time. The developer loads the program into the emulator and can then run, step and trace into it, much like it is done on PC:s. Many emulators have more advanced features like performance analysis, coverage analysis, a trace buffer and advanced trigger and breakpoint possibilities.
Inside the ICE, or usually on the pod, is a processor of the kind the emulator replaces, or a special bond-out version of the same chip. Bond-out chips have normally internal signals and/or busses bonded out to its legs. This is in order to let the ICE, and the developer, get a more complete picture of the status of the chip. Often emulators that use bond-out chips have more features then those that don't.



An ICE is just one of the many debugging tools at your disposal. It's also among the most powerful.

Embedded systems pose unique debugging challenges. With neither terminal nor display (in most cases), there's no natural way to probe these devices, to extract the behavioral information needed to find what's wrong.
An in-circuit emulator (ICE) is one of the oldest embedded debugging tools, and is still unmatched in power and capability. It is the only tool that substitutes its own internal processor for the one in your target system. Using one of a number of hardware tricks, the emulator can monitor everything that goes on in this on-board CPU, giving you complete visibility into the target code's operation. In a sense, the emulator is a bridge between your target and your workstation, giving you an interactive terminal peering deeply into the target and a rich set of debugging resources.

Until just a few years ago, most emulators physically replaced the target processor. Users extracted the CPU from its socket, plugging the emulator's cable in instead. Today, we're usually faced with a soldered-in surface-mounted CPU, making connection strategies more difficult. Some emulators come with an adapter that clips over the surface-mount processor, tri-stating the device's core, and replacing it with the emulator's own CPU. In other cases, the

emulator vendor provides adapters that can be soldered in place of the target CPU. As chip sizes and lead pitches shrink, the range of connection approaches expands.

Target access

An emulator's most fundamental resource is target access: the ability to examine and change the contents of registers, memory, and I/O. However, since the ICE replaces the CPU, it generally does not need working hardware to provide this capability. This makes the ICE, by far, the best tool for troubleshooting new or defective systems. For example, you can repeatedly access a single byte of RAM or ROM, creating a known and consistent stimulus to the system that is easy to track using an oscilloscope.

Breakpoints are another important debugging resource. They give you the ability to stop your program at precise locations or conditions (like "stop just before executing line 51"). Emulators also use breakpoints to implement single stepping, since the processor's single-step mode, if any, isn't particularly useful for stepping through C code.

There's an important distinction between the two types of breakpoints used by different sorts of debuggers. Software breakpoints work by replacing the destination instruction by a software interrupt, or trap, instruction. Clearly, it's impossible to debug code in ROM with software breakpoints. Emulators generally also offer some number of hardware breakpoints, which use the unit's internal magic to compare the break condition against the execution stream. Hardware breakpoints work in RAM or ROM/flash, or even unused regions of the processor's address spaces.
Complex breakpoints let us ask deeper questions of the tool. A typical condition might be: "Break if the program writes 0x1234 to variable buffer, but only if function get_data() was called first." Some software-only debuggers (like the one included with Visual C++) offer similar power, but interpret the program at a snail's pace while watching for the trigger condition. Emulators implement complex breakpoints in hardware and, therefore, impose (in general) no performance penalty.

ROM and, to some extent, flash add to debugging difficulties. During a typical debug session we might want to recompile and download code many times an hour. You can't do that with ROM. An ICE's emulation memory is high-speed RAM, located inside of the emulator itself, that maps logically in place of your system's ROM. With that in place, you can download firmware changes at will.
Many ICEs have programmable guard conditions for accesses to both the emulation and target memory. Thus, it's easy to break when, say, the code wanders off and tries to write to program space, or attempts any sort of access to unused addresses.

Nothing prevents you from mapping emulation memory in place of your entire address space, so you can actually debug much of the code with no working hardware. Why wait for the designers to finish? They'll likely be late anyway. Operate the emulator in stand-alone mode (without the target) and start debugging code long before engineering delivers prototypes.

Real-time trace is one of the most important emulator features, and practically unique to this class of debugging tool. Trace captures a snapshot of your executing code to a very large memory array, called the trace buffer, at full speed. It saves thousands to hundreds of thousands of machine cycles, displaying the addresses, the instructions, and transferred data.

The emulator and its supporting software translates raw machine cycles to assembly code or even C/C++ statements, drawing on your source files and the link map for assistance.
Trace is always accompanied by sophisticated triggering mechanisms. It's easy to start and stop trace collection based on what the program does. An example might be to capture every instance of the execution of an infrequent interrupt service routine. You'll see everything the ISR does, with no impact on the real time performance of the code.
Generally, emulators use no target resources. They don't eat your stack space, memory, or affect the code's execution speed. This "non-intrusive" aspect is critical for dealing with real-time systems.

Practical realities

Be aware, though, that emulators face challenges that could change the nature of their market and the tools themselves. As processors shrink, it gets awfully hard to connect anything to those whisker-thin package leads. ICE vendors offer all sorts of adapter options, some of which work better than others.
Skyrocketing CPU speeds also create profound difficulties. At 100MHz, each machine cycle lasts a mere 10ns; even an 18-inch cable between your target and the ICE starts to act as a complex electrical circuit rather than a simple wire. One solution is to shrink the emulator, putting all or most of the unit nearer the target socket. As speeds increase, though, even this option faces tough electrical problems.
Skim through the ads in Embedded Systems Programming and you'll find a wide range of emulators for 8- and 16-bit processors, the arena where speeds are more tractable. Few emulators exist, though, for higher-end processors, due to the immense cost of providing fast emulation memory and a reliable yet speedy connection.
Oddly, one of the biggest user complaints about emulators is their complexity of use. Too many developers never use any but the most basic ICE features. Sophisticated triggering and breakpoint capabilities invariably require rather complex setup steps. Figure on reading the manual and experimenting a bit. Such up-front time will pay off later in the project. Time spent in learning tools always gets the job done faster.
Based on ***Introduction to In-Circuit Emulators*** by Jack Ganssle, Embedded Systems Programming (11/30/01, 11:22:36 AM EDT)

Exercise description

During the laboratory students are to write assembler program given by the supervisor. The program should be assembled and loaded into the ICE environment. Students are to debug and run the program using ICE.

Software tools for writing the program:

- ConText editor
- ASM51 assembler with ASM51.PDF documentation
- Signum Systems emulator software

Hardware tools for debugging:

- USP51 In-Circuit Emulator

Important informations:

- Basic program template is located on **D:\Lab_emu\emu.asm**
- Assembling is executed on User Command No.1 in ConText editor (head with number one in button's menu)
- Shortcut to program for emulator is located on the desktop
- There are double-coloured LED connected to the test board, it is connected to the port P1.2 and P1.3 of the microcontroller. Write `clr P1.x` to turn the LED on, `setb P1.x` to turn the LED off.

The report should contain:

- Title page
- Topic of the laboratory
- Program with comments
- Conclusions