DESIGNING WITH COMPUTING PLATFORMS

System Architecture

We know that architecture is a set of elements and the relationships between them that together form a single unit. The architecture of an embedded computing system is the blueprint for implementing that system it tells you what components you need and how you put them together. The architecture of an embedded computing system includes both hardware and software elements. Let's consider each in turn. The hardware architecture of an embedded computing system is the more obvious manifestation of the architecture since you can touch it and feel it. It includes several elements, some of which may be less obvious than others.

CPU An embedded computing system clearly contains a microprocessor. But which one? There are many different architectures, and even within an architecture we can select between models that vary in clock speed, bus data width, integrated peripherals, and so on. The choice of the CPU is one of the most important, but it cannot be made without considering the software that will execute on the machine.

Bus The choice of a bus is closely tied to that of a CPU, since the bus is an integral part of the microprocessor. But in applications that make intensive use of the bus due to I/O or other data traffic, the bus may be more of a limiting factor than the CPU. Attention must be paid to the required data bandwidths to be sure that the bus can handle the traffic.

Memory Once again, the question is not whether the system will have memory but the characteristics of that memory. The most obvious characteristic is total size, which depends on both the required data volume and the size of the program instructions. The ratio of ROM to RAM and selection of DRAM versus SRAM can have a significant influence on the cost of the system. The speed of the memory will play a large part in determining system performance.

Input and output devices The user's view of the input and output mechanisms may not correspond to the devices connected to the microprocessor.

For example, a set of switches and knobs on a front panel may all be controlled by a single microcontroller, which is in turn connected to the main CPU. For a given function, there may be several different devices of varying sophistication and cost that can do the job. The

difficulty of using a particular device, such as the amount of glue logic required to interface it, may also play a role in final device selection.

Hardware Design

The design complexity of the hardware platform can vary greatly, from a totally off-the-shelf solution to a highly customized design. At the board level, the first step is to consider **evaluation boards** supplied by the microprocessor manufacturer or another company working in collaboration with the manufacturer. Evaluation boards are sold for many microprocessor systems; they typically include the CPU, some memory, a serial link for downloading programs, and some minimal number of I/O devices. Figure 1.7.1 shows an ARM evaluation board manufactured by Sharp.

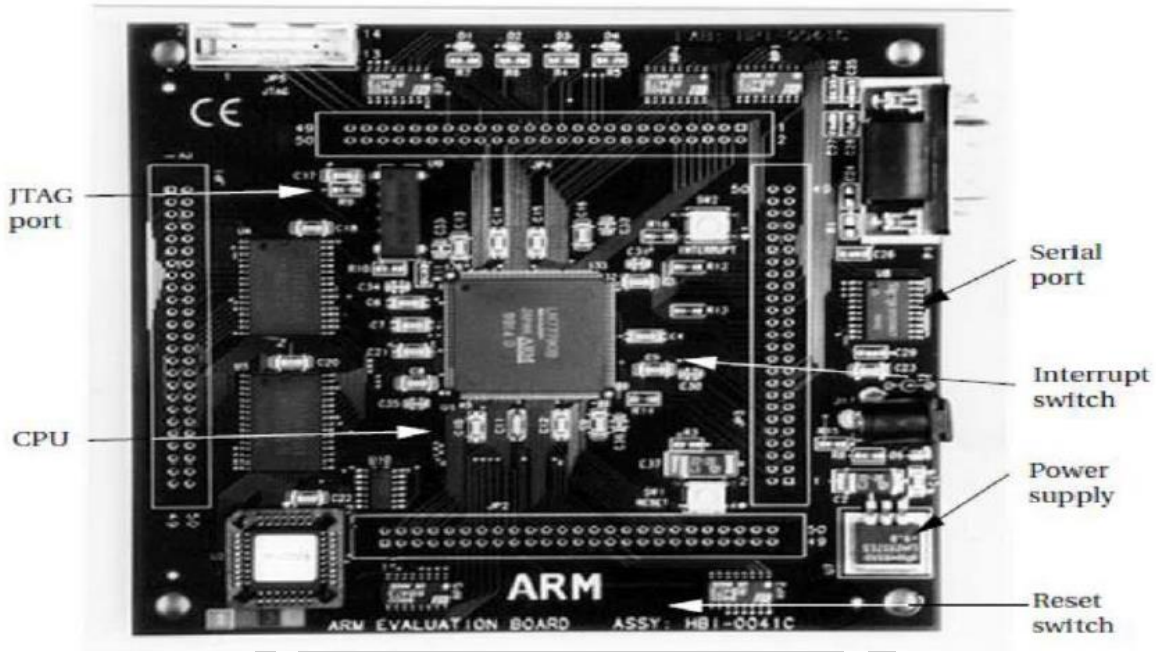

Figure 1.7.1 An ARM evaluation board

[Source: Embedded Systems: An Integrated Approach by Lyla B. Das]

The evaluation board may be a complete solution or provide what you need with only slight modifications. If the evaluation board is supplied by the microprocessor vendor, its design (netlist, board layout, etc.) may be available from the vendor; companies provide such information to make it easy for customers to use their microprocessors. If the evaluation board comes from a third party, it may be possible to contract them to design a new board with your required modifications, or you can start from scratch on a new board design. The other major task is the choice of memory and peripheral components. In the case of I/O devices, there are

two alternatives for each device: selecting a component from a catalog or designing one yourself. When shopping for devices from a catalog, it is important to read data sheets carefully it may not be trivial to figure out whether the device does what you need it to do.

Development Environments

A typical embedded computing system has a relatively small amount of everything, including CPU horsepower, memory, I/O devices, and so forth. As a result, it is common to do at least part of the software development on a PC or workstation known as a **host** as illustrated in Figure The hardware on which the code will finally run is known as the **target**. The host and target are frequently connected by a USB link, but a higher-speed link such as Ethernet can also be used.

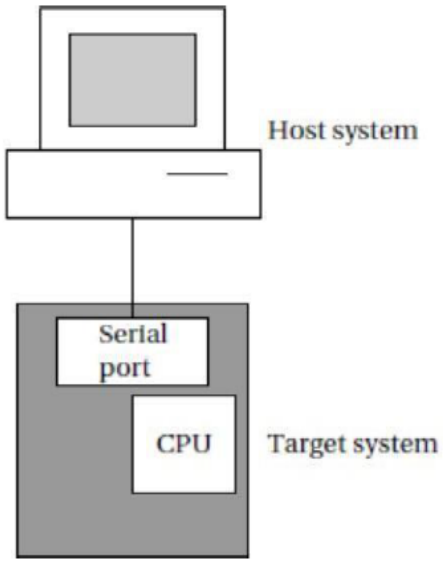

Figure 1.7.2 Connecting a host and a target system

[Source: Embedded Systems: An Integrated Approach by Lyla B. Das]

The target must include a small amount of software to talk to the host system. That software will take up some memory, interrupt vectors, and so on, but it should generally leave the smallest possible footprint in the target to avoid interfering with the application software. The host should be able to do the following:

- load programs into the target,
- start and stop program execution on the target, and
- examine memory and CPU registers

Debugging Techniques:

A good deal of software debugging can be done by compiling and executing the code on a PC or workstation. But at some point it inevitably becomes necessary to run code on the embedded hardware platform. Embedded systems are usually less friendly programming environments than PCs. Nonetheless, the resourceful designer has several options available for debugging the system. The serial port found on most evaluation boards is one of the most important debugging tools. In fact, it is often a good idea to design a serial port into an embedded system even if it will not be used in the final product; the serial port can be used not only for development debugging but also for diagnosing problems in the field. Another very important debugging tool is the **breakpoint**.

The simplest form of a breakpoint is for the user to specify an address at which the program's execution is to break. When the PC reaches that address, control is returned to the monitor program. From the monitor program, the user can examine and/or modify CPU registers, after which execution can be continued. Implementing breakpoints does not require using exceptions or external devices.

Debugging Challenges

Logical errors in software can be hard to track down, but errors in real-time code can create problems that are even harder to diagnose. Real-time programs are required to finish their work within a certain amount of time; if they run too long, they can create very unexpected behavior.

The exact results of missing real-time deadlines depend on the detailed characteristics of the I/O devices and the nature of the timing violation. This makes debugging real-time problems especially difficult. Unfortunately, the best advice is that if a system exhibits truly unusual behavior, missed deadlines should be suspected. In-circuit emulators, logic analyzers, and even LEDs can be useful tools in checking the execution time of real-time code to determine whether it in fact meets its deadline.