What:	• A broad handbook chapter on microprocessors in electronic instruments
Goal:	• To provide the most needed information for the average reader • How the instrument is built and works • Sense of where things are heading with embedded computers in instruments • Implications for instrument usefulness given the microprocessor design
Reader:	• Beginning user • Student • Experienced practitioner • Instrument evaluator/selector • Not targeted at an instrument designer.

Chapter 10: Embedded Computers in Electronic Instruments ... 3
10.1 Introduction ... 3
10.2 Embedded Computers ... 3
10.2.1 Embedded Computer Model ... 4
10.2.2 Embedded Computer Uses ... 4
10.2.3 Benefits of embedded computers in instruments ... 6
10.3 Embedded Computer System Hardware ... 7
10.3.1 Microprocessors as the heart of the Embedded Computer ... 7
10.3.2 How Microprocessors Work ... 7
10.3.3 Program and Data Store ... 7
10.3.4 Machine instructions ... 8
10.3.5 Integer and floating point instructions ... 9
10.3.6 Internal registers ... 9
10.3.7 Interrupts ... 9
10.3.7 Cache ... 10
10.3.8 RISC versus CISC ... 10
10.4 Elements of an embedded computer ... 10
10.4.1 Support circuitry ... 10
10.4.2 Memory ... 10
10.4.3 Non-volatile memory ... 11
10.4.4 Peripheral components ... 11
10.4.5 Timers ... 12
10.4.6 Instrument hardware ... 12
10.5 Physical form of the embedded computer ... 12
10.6 Architecture of the embedded computer instrument ... 13
10.7 Embedded Computer System Software ... 16
10.7.1 How Embedded Computers are programmed ... 16
10.7.2 Assembly Language Development ... 16
10.7.3 High-Level Language Development ... 18
10.7.4 High-level language compilers ... 19
10.7.5 Operating systems software ... 20
10.7.6 Real Time ... 22
10.8 User Interfaces ... 22
10.9 External Interfaces ... 23
10.9.1 Hardware interface characteristics ... 24
10.9.2 Hardware interface standards ... 25
10.9.3 Software protocol standards ... 26
10.10 Numerical issues ... 27
10.10.1 Integers ... 27
10.10.2 Floating Point Numbers ... 28
10.10.3 Scaling and fixed point representations ... 29
10.10.4 Big-endian and Little-endian ... 30
10.11 Instrumentation Calibration and Correction Using Embedded Computers ... 31
10.11.1 Calibration ... 31
10.11.2 Linear Calibration ... 32
10.11.3 Correction ... 32
10.12 Using instruments that contain embedded computers ... 32
10.12.1 Instrument customization ... 33
10.12.2 User access to an embedded computer ... 34
10.12.3 Environmental considerations ... 35
10.12.4 Instrument Longevity ... 36

Chapter 10: Embedded Computers in Electronic Instruments

10.1 Introduction

All but the simplest electronic instruments have some form of embedded computer system. Given this, it is important in a handbook on electronic instruments to provide some foundation on embedded computers. The goals of this chapter[1] are to describe:

- What embedded computers are.
- How embedded computers work.
- What embedded computers are used for in instruments.

The *embedded computer* is exactly what the name implies – it is a computer put into (i.e. embedded) in a device. The goal of the device is not to be a general-purpose computer, but to provide some other function. In the focus of this chapter and book, the devices we are talking about are instruments.

Embedded computers are almost always built from microprocessors or microcontrollers. *Microprocessors* are the physical hardware *integrated circuit* (IC) that is the *central processing unit* (CPU) of the computer. In the beginning, microprocessors were miniature, simplified versions of larger computer systems. Since they were smaller versions of their larger relatives, they were dubbed *micro*processors. *Microcontrollers* are a single IC with the CPU and the additional circuitry and memory to make an entire embedded computer.

In the context of this chapter, embedded computer refers to the full, embedded computer system used within an instrument. Microprocessor and microcontroller refer to the IC hardware (i.e. the chip).

10.2 Embedded Computers

Originally, there were no embedded computers in electronic instruments. The instruments consisted of the raw analog and (eventually) digital electronics. As computers and digital electronics advanced, instruments began to add limited connections to external computers. This dramatically extended what a user could do with the instrument (involving calculation, automation, ease of use and integration into systems of instruments). With the advent of microprocessors, there was a transition to some computing along with the raw measurement inside the instrument – embedded computing.

There is a shift to more and more computing embedded in the instrument because of reductions of computing cost and size, increasing computing power and increasing number of uses for computing in the instrument domain. Systems that were previously a computer or PC and an instrument are now just an instrument. (In fact, what used to take five bays of six foot high equipment racks including a minicomputer is now in a 8"x18"x20" instrument.) So, more and more computing is moving into the instrument.

This transition is happening due to the demand for functionality, performance, and flexibility in instruments and also due the low cost of microprocessors. In fact, the cost of microprocessors is sufficiently low and their value is sufficiently high that most instruments have more than one embedded computer. There is

[1] This chapter includes material developed from Joe Mueller's chapter "Microprocessors in Electronic Instruments" from the second edition of this handbook.

also a certain amount of the inverse happening - instrumentation going into personal computers in the form of plug-in boards. In many of these plug-in boards, there are still embedded computers.

10.2.1 Embedded Computer Model

The instrument and its embedded computer normally interact with four areas of the world: the measurement, the user, peripherals and external computers. The instrument needs to take in **measurement** input and/or send out **source** output. A *source* is defined as an instrument that generates or synthesizes signal output. An *analyzer* is defined as an instrument that analyzes or measures input signals. These signals can consist of analog and/or digital signals. (Note that throughout this chapter measurement means both input analysis and output synthesis/generation of signals.) The *front end* of the instrument is the portion of the instrument that conditions, shapes or modifies the signal to make it suitable for acquisition by the analog to digital converter. The instrument normally interacts with the **user** of the measurement. The instrument also generally interacts with an **external computer**, which is connected for control or data connectivity purposes. Finally, in some cases, the instrument is connected to **local peripherals,** primarily for printing and storage. The following figure (Figure 10.1) shows a generalized block diagram the embedded computers and these aspects.

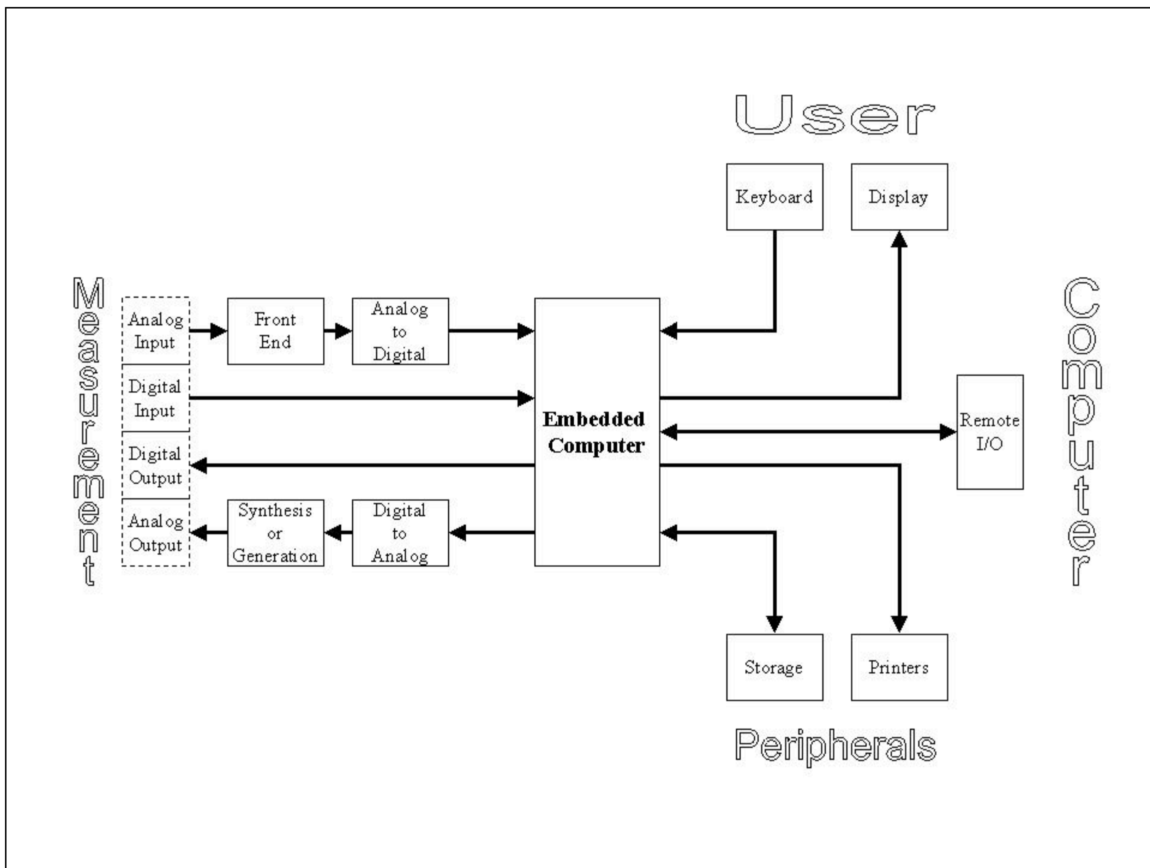

Figure 10.1: Embedded Computer Generalized Block Diagram

10.2.2 Embedded Computer Uses

The embedded computer, as seen from the previous discussion, has taken a central role in the operation and function of an instrument. Embedded computers have a wide range of specific uses (related to the measurement, user, external computer and local peripheral aspects) within the instrument:

- External **computer** interfaces
- **User** interfaces
- **User** configuration and customization
- **User**-defined automation
- **Measurement** or source calculation
- **Measurement** or source control
- **Measurement** or source calibration and self tests
- **Measurement** or source error and failure notification
- Local **peripheral** control
- Coordination of the various tasks

The following table (Table 10.1) describes these uses.

Table 10.1: Uses of embedded computers

Use	Description
• External **computer** interfaces	The embedded computer is typically involved in the control and transfer of data through external interfaces. This allows for the connection of the instrument to external PCs, networks and peripherals. Examples, which are described later in this chapter, include IEEE 488 (also known as GPIB or HP-IB), RS-232 (serial), Centronics (parallel), Universal Serial Bus (USB), IEEE 1394 (FireWire), et cetera.
• **User** interfaces	The embedded computer is also typically involved in the display to and input from the user. Examples include keyboards, switches, rotary pulse generators (RPGs - i.e. knobs), Light Emitting Diodes (LEDs – single or alpha-numeric displays), Liquid Crystal Displays (LCDs), CRTs, touch screens, …
• **User** configuration and customization	Many instruments often have a large amount of configuration information due to their advanced capabilities. The embedded computer enables saving and recalling of the instrument state. Also, the embedded computer sometimes is used for user customization of the instrument. This can range from simple configuration modifications through complete instrument programmability.
• **User**-defined automation	With very powerful embedded computers available in instruments, it is often unnecessary to connect the instrument to an external computer for more advanced tasks. Examples include go/no-go (also known as pass/fail) testing and data logging.
• **Measurement** or source calculation	The embedded computer almost always does calculations, ranging from very simple to very complex, that convert the raw measurement data to the target instrument information for measurement or vice versa for source instruments. For example, electrical transducers like thermocouples don't produce results that are in the terms that the users want. Embedded computers do the calculations to convert the measured voltage to the desired temperature reading.
• **Measurement** or source control	The embedded computer generally controls the actual measurement process. This can include control of a range of functions like Analog to Digital conversion, switching, filtering, detection, shaping and so on. Note that this is not the analog or digital electronics, but the control of these components that do the measuring or synthesizing.
• **Measurement** or source calibration and self tests	Many instruments are very complex. The embedded computer is almost always used to do at least a small amount of self-test. Most instruments

	use embedded computers for more extensive calibration tests.
• **Measurement** or source error and failure notification	As with calibration, many instruments are very complex – not only in the basic hardware and system but also in the type of measurements. Instruments often do sampling and statistical analysis. Acceptable measurement data can be refined and verified with a microprocessor in the system. Notification of marginal, out of bounds or failure conditions can be given.
• Local **peripheral** control	Many instruments have built in storage devices. These are often floppy disk drives. There are sometimes built-in printers, but more often there is a connection to an external, but local, printer. Less common are local measurement-related peripherals like switches or attenuators that are controlled by the embedded computer.
• Coordination of the various tasks	The previous uses interact in various ways. A key use of embedded computers is to organize, synchronize and control the various aspects of the instrument.

10.2.3 Benefits of embedded computers in instruments

In addition to the direct uses of embedded computers, it is instructive to think about the value of an embedded computer inside an instrument. The benefits occur throughout the full lifecycle of an instrument from development through maintenance. These benefits are described in the following table (Table 10.2).

Table 10.2: Benefits of embedded computers through its lifecycle

Lifecycle phase	Benefit of embedded computers
• Development	One of the biggest advantages of embedding computers inside an instrument is that they allow several aspects of the hardware design to be simplified. In many instruments, the embedded computer participates in acquisition of the measurement data by servicing the measurement hardware. Embedded computers also simplify the digital design by providing mathematical and logical manipulations, which would otherwise be done in hardware. They also provide calibration both through numerical manipulation of data and by controlling calibration hardware. This is the classic transition of function from hardware to software.
• Manufacturing	The embedded computer allows for lower manufacturing costs through effective automated testing of the instrument. Embedded computers also are a benefit since they allow for easier and lower cost defect fixes and upgrades (with a ROM or program change).
• Installation	When used as a stand-alone instrument, embedded computers can make the set-up much easier by providing on-line help or set-up menus. This also includes automatic or user-assisted calibration. Although many are stand-alone instruments, a large number are part of a larger system. The embedded computers often make it easier to connect an instrument to a computer system by providing multiple interfaces and simplified or automatic set up of interface characteristics.
• Use	Given the complexity of many instruments, it becomes more difficult to coherently present functionality to the user. The embedded computers allow for user interfaces that are easier to learn and use. This also applies to local language, data, and numerical format customization of the instrument.
• Maintenance	Given the complexity of function, it is often difficult to tell if the instrument is operating properly. The embedded computer allows for a system that will check itself and its measurement hardware at various

	times (at power-on, periodically, before each measurement) and describe not only the presence of a failure, but also repair suggestions.

10.3 Embedded Computer System Hardware

10.3.1 Microprocessors as the heart of the Embedded Computer

The embedded computer in an instrument requires both hardware and software. The microprocessor is just one of the hardware components. A full embedded computer also requires support circuitry, memory and peripherals (including the instrument hardware). The microprocessor provides the Central Processing Unit (CPU) of the embedded computer. Some microprocessors are very complex while others are fairly rudimentary. There is variation in the amount of integration of functionality onto microprocessors – this can include memory, I/O and support circuitry.

10.3.2 How Microprocessors Work

A microprocessor is basically a state machine that goes through various state changes determined by its program and its external inputs. This program is a list of machine language instructions that are stored in memory. The microprocessor accesses the program by generating a memory storage location or address on the *address bus*. The memory then returns the appropriate instruction (or data) from that location over the *data bus*. The machine-language instructions are numbers that are returned to the microprocessor.

10.3.3 Program and Data Store

The address bus and the data bus are physical connections between the microprocessor and memory. The number of connections on each bus varies and has grown over time. In a high-level architectural sense, there are two general classes of memory - program store and the data store. The *program store* is, as the name implies, the memory where the program is stored. Similarly, the *data store* is where data values used by the program are stored and read. The general structure of the microprocessor connected to program and data store can be seen in the following figure (Figure 10.2). In most cases, the *input/output* (I/O) devices are connected via this same address and data bus mechanism.

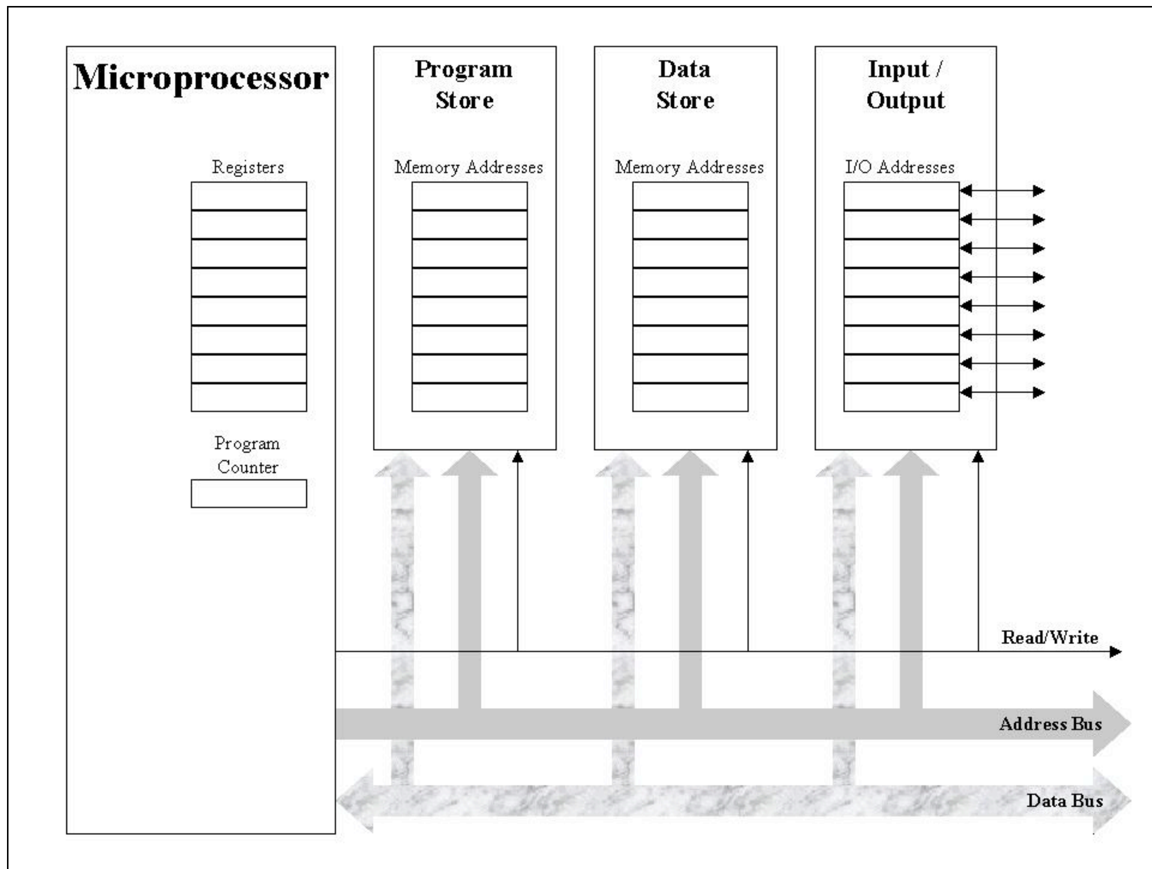

Figure 10.2: Microprocessor Basic Block Diagram

The data store is very similar to the program store. It has two primary functions: (1) It is used to store values that are used in calculations. (2) It also provides the microprocessor with a subroutine stack. The *read/write* line shown in Figure 10.2 is the signal used by the data store memory to indicate a memory read or a memory write operation. Depending on the microprocessor system, there are other control lines beyond the scope of this chapter used to control the memory devices and address and data bus. The subroutine stack is an integral part of the computer that provides for subroutine call and return capability. The *return address* (the next address for the instruction pointer after the call instruction) is pushed onto the stack as the call instruction is executed so that when the return instruction is executed it gets the appropriate next instruction address. Calls generally push the instruction pointer onto the stack and it is the job of the developer or the high-level language compiler to deal with saving registers. Pushing information onto the stack is one of the common ways to pass parameters to the subroutine. For interrupts, the information pushed on the stack often includes not only the return address information, but also various microprocessor registers.

Although the data store needs to be writable, there does not need to be a physical difference between program store and data store memory. Therefore, most microprocessors do not differentiate between program and data. Microprocessors like this are referred to as *Von Neumann* machines and have program and data in the same memory space. There are also quite a few microprocessors that do have separate program and data space (inside or outside of the microprocessor) called *Harvard* machines. This differentiated memory is useful because it allows for simpler and/or faster CPU design.

10.3.4 Machine instructions

The machine language instructions indicate to the microprocessor what sort of actions it should take. The following table (Table 10.3) shows examples of the types of operations that a microprocessor may support.

Table 10.3: Types of microprocessor operations

Type of operation	Examples
• Data	load, store, clear
• Stack	push, pop
• Integer math	add, subtract, multiply, divide
• Boolean logic	and, or, not, nor, xor, shift, rotate
• Branching	comparisons, conditional branch, unconditional branch
• Subroutine	call, return
• Floating point math	add, subtract, multiply, divide
• Interrupts	enable, disable, generate interrupt

Microprocessors compete on the richness of their instruction set, data width, address space and their speed. Therefore, there is tremendous variety in the instructions and how they operate.

10.3.5 Integer and floating point instructions

All microprocessors have integer mathematical operations, but many do not have floating point operations built in. For microprocessors with no floating point facilities, the developer is required to use software routines, which are very costly in time, to handle floating point operations. Some microprocessors have pre-defined floating point operations, but without built-in floating point hardware. In these cases, when the floating point instructions are encountered, the microprocessor will generate internal signals that cause an optional floating point coprocessor to perform the operation. If there is no coprocessor present, a CPU exception or trap is generated that causes a software emulation of the floating point operation to occur. The software emulation is slower than hardware floating point but it provides the advantage that the software can be identical across a range of microprocessor hardware.

10.3.6 Internal registers

Microprocessors also have various internal registers. Common registers store the instruction address (also called the instruction pointer or program counter register), data addresses and data. There is a wide variety in the number of general address and data registers. For more advanced microprocessors, there are both integer and floating point registers. Microprocessors also have a variety of status registers available to the programmer.

10.3.7 Interrupts

Another critical part of microprocessor architecture is the interrupt. The concept is that a microprocessor is executing a program and it needs to respond to an important event. An *interrupt* is a hardware signal sent to the microprocessor that an important has occurred. The interrupt forces the microprocessor at the end of the current instruction to do a subroutine call to an *interrupt service routine*. The microprocessor performs the instructions to handle the event and then returns to the previous program. Although this is a critical part of computers used in general applications, it is especially important in instrument applications. Interrupts can be the trigger of a measurement, data transfer completion, error conditions, user input, and so on. Just as instruction sets vary, the interrupt system design of a microprocessor can very greatly from a single interrupt approach to multiple interrupts with various priorities or levels.

10.3.7 Cache

In many current and high-performance microprocessors, there is a feature called cache. This is a portion of high speed memory that the microprocessor uses to store copies of frequently used memory locations. This is helpful because the main semiconductor memory of computer systems is often slower (often around 100 nanoseconds access time) than the microprocessor can access and use memory. To help with this imbalance, cache memory stores or pre-stores memory locations for these frequently used areas, which could be either instructions (in *instruction cache*) or data (in *data cache*). Cache tends to be in the 10 nanosecond access time range. The cache memory can be located on the microprocessor and is called *primary cache* memory (usually less than 64 Kbytes of memory). The cache memory can also be located outside of the microprocessor. This external cache is called *secondary cache* memory (usually in the 256 Kbytes to 512 Kbytes range). Cache has a huge performance benefit – especially in loops and common routines that fit in cache. It also can represent a challenge for the instrument designer because cache causes the microprocessor to operate non-deterministically with respect to performance – it can run faster or slower depending on the data stream and inputs to the computer system.

Some microprocessors have internal bus widths that are different than what is brought out to the external pins. This is done to allow for a faster processor at a lower cost. In these cases, the external (to the microprocessor) data paths will be narrower (e.g. 16 data bits wide) while the internal microprocessor data paths will be full width (e.g. 32 data bits wide). When the microprocessor gets data into or out of a full width register it would perform two sequential read or write operations to get the full information. This is especially effective when coupled with caching. This allows for segments of code or loop structures to operate very efficiently (on the full width data) even though the external implementation is less expensive partial width hardware.

10.3.8 RISC versus CISC

Another innovation in computing that has been making its way into instruments is microprocessors based on *Reduced Instruction Set Computers* (RISC). This is based on research that shows that a computer system can be designed to operate more efficiently if all of its instructions are very simple and they execute in a single clock cycle. This is different from the classical *Complex Instruction Set Computer* (CISC) model. Chips like the Intel x86 and Pentium families and the Motorola 68000 family are CISC microprocessors. Chips like the Motorola PowerPC and joint HP and Intel IA-64 microprocessors are RISC systems. One of the challenges for instrument designers is that RISC systems usually have a very convoluted instruction set. Most development for RISC systems require advanced compiler technologies to achieve high performance and allow developers to easily use them. Another characteristic of RISC systems is that their application code tends to be larger than CISC systems because the instruction set is simpler.

10.4 Elements of an embedded computer

10.4.1 Support circuitry

Although requirements vary, most microprocessors require a certain amount of support circuitry. This includes the generation of a system clock, initialization hardware, and bus management. In a conventional design, this often takes 2 or 3 external integrated circuits (ICs) and 5 to 10 discrete components. The detail of the design at this level depends heavily on the microprocessor used. In complex or high volume designs, an Application-Specific Integrated Circuit (ASIC) may be used to provide much of this circuitry.

10.4.2 Memory

The microprocessor requires memory both for program and data store. Embedded computer systems usually have both ROM and RAM. *Read Only Memory* (ROM) is memory whose contents do not change even if power is no longer applied to the memory. *Random Access Memory* (RAM) is a historical, but inadequate term, that really refers to read/write memory - memory whose contents can be changed.

RAM memory is volatile – it will loose its contents when power is no longer applied. RAM is normally implemented as either *static* or *dynamic* devices. Static memory is a type of electrical circuit[2] that will retain its data with or without access as long as power is supplied. Dynamic memory is built out of a special type circuit[3] that requires periodic memory access (every few milliseconds) to refresh and maintain the memory state. This is handled by memory controllers and requires no special attention by the developer. The advantage of the dynamic memory RAM is that it consumes much less power and space.

ROM is used for program storage because the program does not usually change after power is supplied to the instrument. There are a variety of technologies used for ROM in embedded applications:

- Mask ROM – custom programmed at the time of manufacture, unchangeable
- Fusible link Programmable ROM (PROM) – custom programmed before use, unchangeable after programming
- Erasable PROM (EPROM) - custom programmed before use, can be reprogrammed after erasing with ultraviolet light
- Electronically Erasable PROM (EEPROM) – custom programmed before use, can be reprogrammed after erasing - like an EPROM but the erasing is done electrically.

10.4.3 Non-volatile memory

Some instruments are designed with special nonvolatile RAM; i.e. memory that maintains its contents after power has been removed. This is necessary for storing information like calibration and configuration data. This can be implemented with regular RAM memory that has a battery backup. It can also be provided by special nonvolatile memory components – most commonly *flash memory* devices. Flash memory is a special type of EEPROM that uses block transfers (instead of individual bytes) and has a fairly slow (in computer terms) write time. So, it is not useful as a general read/write memory device, but is perfect for non-volatile memory purposes. There are also a limited number of writes allowed (on the order of 10,000).

All embedded systems will have either a ROM/RAM or a flash/RAM memory set so that the system will be able to operate the next time the power is turned on.

10.4.4 Peripheral components

Microprocessors normally have several peripheral components. These are usually *Very Large Scale Integration* (VLSI) components that provide some major functionality. These peripheral components tend to have a small block of registers or memory that control their hardware functions.

For example, a very common peripheral component is a *Universal Asynchronous Receiver/Transmitter* (UART). It provides a serial interface between the instrument and an external device or computer. UARTs have registers for configuration information (like data rate, number of data bits, parity) and for actual data transfers. When the microprocessor writes a character to the data out register, the UART transmits the character, serially. Similarly, when the microprocessor reads the data in register, the UART transfers the current character that has been received. (This does require that the microprocessor checks or knows through interrupts or status registers that there is valid data in the UART.)

---

[2] Static memory is normally built out of latches or flip-flops.

[3] Dynamic memory is normally built out of a stored-charge circuit that uses a switched capacitor for the storage element.

10.4.5 Timers

Another very common hardware feature is the timer. These are used to generate periodic interrupts to the microprocessor. These can be used for triggering periodic operations. They are also used as *watchdog timers*. A watchdog timer helps the embedded computer recover after a non-fatal software failure. These can happen because of static electricity, radiation, random hardware faults or programming faults. The watchdog timer is set up to interrupt and reset the microprocessor after some moderately long period of time (say one second). As long as everything is working properly, the microprocessor will reset the watchdog timer – before it generates an interrupt. If, however, the microprocessor hangs up or freezes, the watchdog timer will generate a reset that returns the system to normal operation.

10.4.6 Instrument hardware

Given that the point of these microprocessors is instrumentation (measurement, analysis, synthesis, switches, et cetera), the microprocessor needs to have access to the actual hardware of the instrument. This instrument hardware is normally accessed by the microprocessor like other peripheral components - i.e. as registers or memory locations.

Microprocessors frequently interface with the instruments' analog circuits using Analog-to-Digital Converters (ADCs) and Digital-to-Analog Converters (DACs). In an analog instrument, the ADC bridges the gap between the analog domain and the digital domain. In many cases, substantial processing is done after the input has been digitized. Increases in the capabilities of ADCs allow the analog input to be digitized closer to the front end of the instrument, allowing a greater portion of the measurement functions to take place in the embedded computer system. This has the advantages of providing greater flexibility and eliminating errors introduced by analog components. Just as ADCs are critical to analog measuring instruments, DACs play an important role in the design of source instruments (like signal generators). They are also very powerful when used together. For example, instruments can have automatic calibration procedures where the embedded computer adjusts an analog circuit with a DAC and measures the analog response with an ADC.

10.5 Physical form of the embedded computer

Embedded computers in instruments take one of three different forms: a separate circuit board, a portion of a circuit board, or a single chip. In the case of a separate circuit board the embedded computer is a **board-level computer** that is a circuit board separate from the rest of the measurement function. For an embedded computer that is a portion of a circuit board there is a **microprocessor** and its associated support circuitry that comprise the embedded computer with at least some portion of the measurement functions on the same circuit board. A single chip embedded computer can be a **microcontroller**, **Digital Signal Processor** or **microprocessor core** with almost all of the support circuitry built into the chip. The following table (Table 10.4) describes these physical form choices in some additional detail:

Table 10.4: Classes of embedded computers

Embedded computer class	Description	Advantages and Disadvantages
Board-level computer	*A board-level computer* is an embedded computer that is built on a circuit board (or sometimes multiple boards) that is separate	• Advantages: The computer is all there. The system is standard (either PC or workstation) and already designed. The system, because it is based on existing computers, has development

	from the rest of the measurement function. These usually include CPU, memory and user I/O. The designer needs to add disk, display, power and the measurement hardware. These can be a specially designed board, an integrated PC motherboard or even a RISC workstation motherboard.	tools, which aid software development. • Disadvantages: Because it is based on a more general-purpose computer board, it usually is larger in size and power. These boards often require a hard disk drive. There is also the potential for Radio Frequency Interference (RFI).
Standard microprocessor	This is a normal, off the shelf, microprocessor (like a Motorola PowerPC or an Intel Pentium). The computer system around the microprocessor must be designed and integrated with the instrument functions.	• Advantages: Because it is being designed specifically for the instrument, it can be tightly integrated into the instrument application. This can allow for less space, cost and power. • Disadvantages: The embedded computer must be both designed and manufactured.
Single chip microcontroller	A microcontroller is a single IC containing almost the entire embedded computer. It typically includes enough ROM and RAM for the application along with several other peripherals and I/O lines that can be used for digital input or output. Microcontrollers frequently include ADC's (analog to digital converters), DAC's (digital to analog converters), and interface support - such as serial ports.	• Advantages: Because it is all there, there is minimal space, cost and power. Almost all of the pins can be used to interface to the instrument system since no pins are necessary for an address bus. Sometimes these lines may be included, but are often shared with other I/O uses. • Disadvantages: By their nature, they tend to be very limited in what applications they can be used in (primarily by the limited ROM and RAM). They are more difficult to develop applications for since they don't bring out address and data bus information which facilitate development tools. This means that they don't have the support that makes it easy to develop firmware with the development systems.

A *Digital Signal Processor* (DSP) is a special type of microcontroller that includes special instructions for digital signal processing allowing it to perform certain types of mathematical operations very efficiently. These math operations are primarily multiply and accumulate (MAC) functions, which are used in filter algorithms. Like the microcontroller, there is reduced space, cost and power. Almost all of the pins can be used to interface to the instrument system.

Microprocessor cores are custom microprocessor IC segments or elements that are used inside custom designed ICs. In this case, the instrument designer has a portion of an ASIC that is the CPU core. The designer can put much of the rest of the system including some analog electronics on the ASIC creating a custom microcontroller. This approach allows for minimum size and power. In very high volumes, the cost can be very low. However, these chips are very tuned to specific applications. They are also generally difficult to develop.

10.6 Architecture of the embedded computer instrument

Just as there a variety of physical forms that an embedded computer can take, there are also several ways to architect an embedded computer instrument. The architecture of the embedded computer can impact the

instrument's cost, performance, functionality, ease of development, style of use and expandability. The range of choices include:

- Peripheral-style instruments (externally attached to a PC)
- PC plug-in instruments (circuit boards inside a PC)
- Single processor instruments
- Multiple processors instruments
- Embedded PC-based instruments (where the embedded computer is a PC)
- Embedded workstation-based instruments

The following table (Table 10.5) describes these architectural choices in some additional detail:

Table 10.5: Architecture choices for embedded computers

Architecture	Description	Advantages and Disadvantages
Peripheral-style instruments	This is a very simple instrument that is designed to always be used with a PC. They usually have a low-end microcontroller and little or no human interface since they are designed to connect to a PC. They generally have very simple system software.	• Advantages: Because they are 'faceless' they have much lower cost, power and size. There is much simpler design because of the simplicity of HW (mostly the measurement part) and simplicity of the embedded SW (because there is very little). There is also a benefit because the software is up to date because the computer is external. The data is already in the PC so exporting it to application programs is easier. • Disadvantages: They require a PC (or laptop). They may not be as portable as a stand-alone instrument. External system and application changes and versions require ongoing updates. The interface to the PC needs to be chosen carefully to prevent obsolescence.
PC plug-in instruments	These are plug in measurement boards that go into a PC. They are similar in many respects and are a variant of peripheral style instruments. The difference is that the interface is the internal PC bus as opposed to an external I/O interface. Plug-in board instruments still have a microprocessor and simple system software.	• Advantages: Similar to the peripheral style instrument, because they are 'faceless', they have much lower cost, power and size. There is much simpler design because of the simplicity of HW (mostly the measurement part) and simplicity of the embedded SW (because there is very little). • Disadvantages: They have to be in a PC. This also presents some challenges from a support point of view – with respect to different PC configurations and reliability. There are ongoing challenges because of changes in PC buses. There are also changes in PC form factors. There have to be ongoing updates to deal with external system and application changes and versions. Another issue is in dealing with radio frequency interference from the PC.
Single processor instruments	The normal microprocessor-based instrument. They have a local human interface and most have some form of remote	• Advantages: They can operate stand-alone and don't require a PC. This helps with portability and with some of the challenges of PC connectivity, operating systems, and

	computer interface.	applications. • <u>Disadvantages</u>: They are costlier because of the user interface, extra power, size, et cetera. There has to be some instrument connectivity mechanism (i.e. software and drivers) to import instrument data into to the PC.
Multiple processors instruments	High-end instruments often include multiple processors for parallel measurement streams or tasks (like measurement, communication, front panel, and computation). There is usually more than one type of operating system in multiple processor instruments – typically with at least one real time operating system (which is described later in this chapter).	• <u>Advantages</u>: The key advantage is performance – there is more processor power dedicated to tasks. The system can be optimized across tasks – for example using simple microcontrollers for I/O intensive operations, normal processor for RAM/ROM intensive operations. There are also some electrical advantages because the processors can be located closer to measurement hardware. • <u>Disadvantages</u>: Multiple processor instruments are more expensive and larger. There has to be some instrument connectivity mechanism (i.e. software and drivers) to import instrument data into to the PC.
Embedded PC-based instruments	A full PC is inside the instrument as the embedded computer. This can be a normal PC motherboard. The embedded PC operating system is usually Windows 95 or NT. Sometimes DOS is used and Windows CE is an emerging technology.	• <u>Advantages</u>: There are standard high level components that can be used for the embedded computer hardware and software. There is no hardware computer development. The hardware tends to be lower cost. There is more computing power because of ongoing improvements in PC performance. There are established and highly productive software development tools. • <u>Disadvantages</u>: A key challenge is dealing with larger and constrained physical form factor. The designer also has to deal with software form factor (software footprint) because a full OS tends to have many things that are not needed in the instrument. The designer has to deal with obsolescence of the motherboard. The designer also has to deal with operating system changes and/or obsolescence. Some instruments use special form-factor variants of PCs, but even though this helps with the form-factor, it has the disadvantage of no longer being a standard motherboard.

There are some instruments that are based on embedded workstations. This is very much like using an embedded PC, but this type of instrument is based on a high-performance workstation. Normally, these are built around a RISC processor and UNIX. The key advantage is very high performance for computationally intensive applications (usually floating point mathematical operations). Like the embedded PC, there are standard high level components that can be used for the hardware and software. There are also established and highly productive software development tools. Like the embedded PC, the key challenge is dealing with larger and constrained physical form factor. Embedded workstations also tend to be more expensive.

10.7 Embedded Computer System Software

As stated earlier, the embedded computer in an instrument requires both hardware and software components. Embedded computer system software includes:

- **Operating system** - the software environment that the (instrument) applications run within.
- **Instrument application** - the software program that performs the instrument functions on the hardware.
- **Support and utility software** - additional software the user of the instrument requires to configure, operate, or maintain the instrument (like reloading or updating system software, saving and restoring configurations, et cetera).

10.7.1 How Embedded Computers are programmed

As mentioned previously, the instructions for the computer are located in program store memory. *Program development* is fundamentally the process that the software developers use to generate the machine language instructions that perform the intended instrument functions.

Some development for embedded computers for instruments has been done in *assembly* language. This is a convenient representation of the machine language of the embedded computer. It still is at the level of the microprocessor, but it is symbolic rather than ones and zeros. The following table (Table 10.6) shows a simple program fragment in assembly language and the resulting machine code (on the right). This program fragment is intended to show reading some character data and processing the character data when a space is encountered.

Table 10.6: Assembly language program example

Assembly language			Machine code
MOVE	(3F),A	; read from port at address 3F	54 3F
AND	#7F,A	; mask off the upper bit	21 7F
COMP	#20,A	; if the input is equal to 20H (a space)	56 20
BEQ	ParseIt	; then go to the parse routine	85 24 35

The resulting machine code for embedded computers is often put into ROM. Because this programming is viewed as less changeable than general software application development it is referred to as *firmware* (*firm*ware being less volatile than *soft*ware).

10.7.2 Assembly Language Development

The assembly language source code is run through an assembler. The *assembler* is a program that translates the assembly language input into the machine language. The assembler is normally run on a *development computer*, normally a PC that is used to create the software for the embedded computer. Note that the development computer is not usually the *target computer* - the embedded computer in the instrument application. The machine code produced by the development computer is for the target computer microprocessor (the processors of the development and target computers are usually different). This machine code is then transferred to the board containing the microprocessor. For smaller, simpler, systems this is often done by transferring a memory image to a *PROM programmer* – a device that writes the image into a PROM. This PROM is then physically inserted onto the board. This process is shown in the following figure (Figure 10.3).

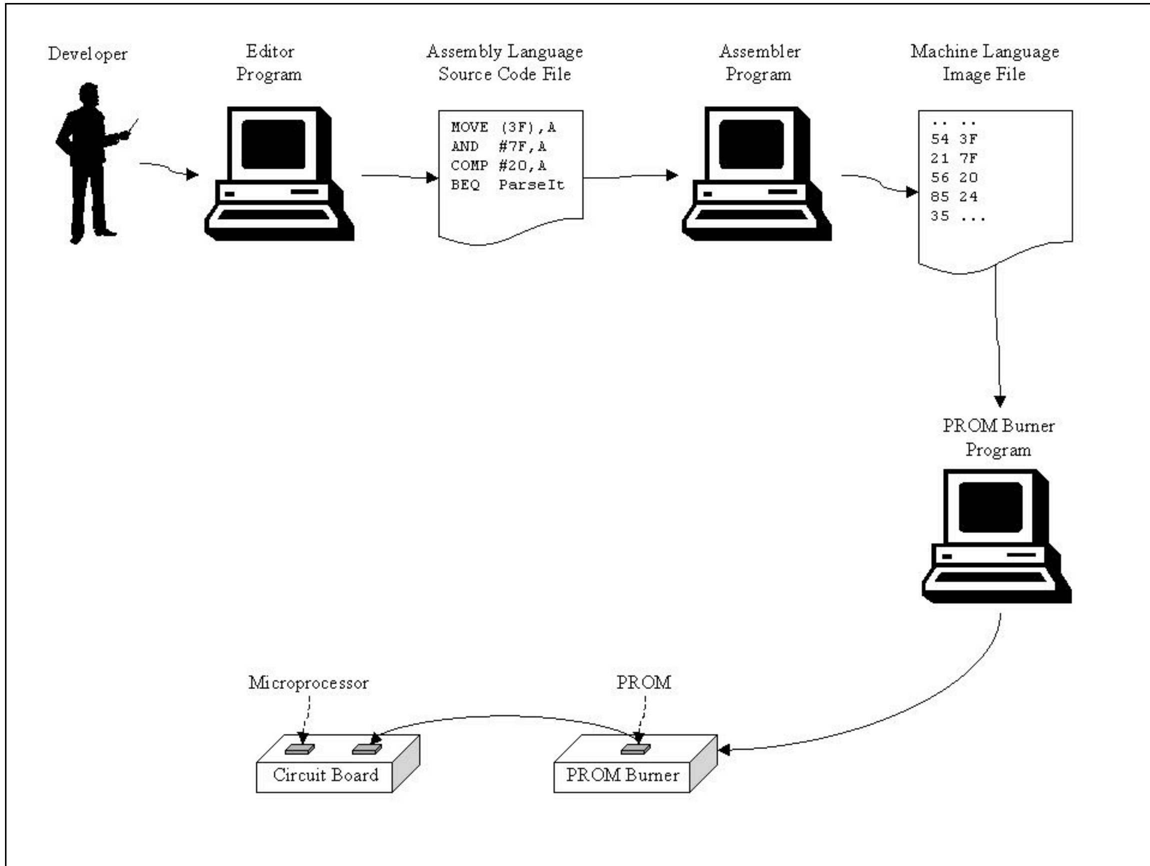

Figure 10.3: Process of assembly language to machine code

This approach of developing code on a development system and placing it into the embedded computer is called *cross development*. As mentioned earlier, there is a fair amount of complexity in embedded computer firmware and software. Because of this complexity, errors are introduced during development. This means that there needs to be tools for debugging the program code. Some of the common tools are described in the following table (Table 10.7).

Table 10.7: Common cross development tools

Tool	Description	Comments
Development boards	Hardware development boards and systems are connected to the microprocessor. They bring out the microprocessor signals. This allows for the developer to see the address lines and the flow of execution. They normally allow for single stepping, limited breakpoints, interrupts and processor resets.	Development boards tend to be very inexpensive but have limited debugging capabilities. They also are not as effective with some classes of microcontrollers - particularly those with limited access to the microprocessor signals.
Monitors	The development prototype of the instrument is built with RAM for program storage and a computer interface – dedicated for development use. The program loaded into the RAM contains some code called a *monitor*. This monitor provides a basic external interface to the	Monitor-based development tends to be inexpensive. The monitors tend to have limited capabilities. They also depend on the microprocessor running properly. If there is a serious hardware or

	instrument over the dedicated interface. The monitor programs generally provide for memory access, starting execution and limited breakpoints.	software problem, the monitor often is not able to work properly.
Logic Analyzers	These are digital logic analyzers built or configured for use with microprocessors. Normally, they are general-purpose logic analyzers with microprocessor pods. The developer attaches the clip onto the microprocessor on the prototype or production printed circuit board. The analyzer will convert address and data line signals into the appropriate code. They generally show some of the other signals as well.	Logic analyzer development is very common. It is a middle ground in terms of cost and functionality between emulators and monitors or development boards.
Emulators	An emulator is a device connected to the instrument being developed in place of the microprocessor. It behaves just like the microprocessor except that the developer has full control over execution: memory access, register access, breakpoints, single stepping, reset, et cetera.	Emulators are very powerful, but are rather expensive. They also tend not to be immediately available for the latest microprocessors.

Emulator development is very powerful, but requires an expensive system for each development station. Monitors or development boards are less expensive but not as powerful. Most major instrument development is done with a combination of these techniques.

As embedded computers have become more powerful and complex, there has been a shift from ROM-based firmware to systems with modifiable program storage memory. This modifiable program storage ranges from Flash memory to integrated hard disk drives. In these systems, the program is loaded into RAM at instrument power up. This has many advantages. It allows the instrument to be modified. It allows for shorter development because there is no step producing the mask-programmable ROMs. It also allows the instrument software to be corrected (or patched) after release. To achieve these characteristics, it is necessary for the embedded computer to include a computer interface and a protocol for downloading new software. And this mechanism needs to be done in a way that the user does not accidentally destroy the instrument software (making for very expensive paperweights).

10.7.3 High-Level Language Development

Another change, as embedded computers have gotten more powerful, is that most instruments are programmed in high-level languages. A *high-level language* is one that is at a higher level of abstraction than assembly or machine language - where the high-level language constructs and capabilities are more complex and conceptual. In a high-level language, the developer writes code to add (e.g. A=B+C). In assembly language the developer has to load a number from a memory location, add a number from a different memory location), and then store the resulting value in a third memory location. This high-level language enables developers to have higher productivity and functionality per line of code written because they are dealing with the application at a level closer to the way they think about it.

There are several high level languages that have and are being used for embedded computing. The most popular languages include C, C++, Java and Pascal. The following table (Table 10.8) compares these languages and assembly language. The example program fragment is based on the same example shown previously with assembly language and is intended to show reading character data and processing the character data when a space is encountered.

Table 10.8: Common embedded computer languages

Language	**Example**	**Comments**

C and C++	if ((PortA & inputMask) == Space)
	  { ParseInput();
	 }
		These are the current popular languages for embedded programming because of space and run-time efficiency. C++ is an object-oriented extension to C. They do not inherently encourage good programming practices.

Assembly	MOVE (3F),A ; read from port 3F AND #7F,A ; mask upper bit COMP #20,A ; if equal to space BEQ ParseIt ; then go parse it	Assembly is not normally used in most instruments (because of the complexity of the software). It is not portable but allows for very high performance and small size because it is symbolic machine code. Often, developers will use a small portion of assembly for critical sections of code with the rest written in a high level language.
Java	Port a = new Port("3F"); if (a.read == ' ') ParseInput();	Java is an object oriented language that compiles to a byte code that is run on a virtual machine. Because of this, it has small executables and is very portable. It does run somewhat slower than classical compiled languages and requires a Java Virtual Machine (JVM) which is often ½ megabyte in memory.

Pascal	CONST port = #3F; CONST space = 32;
	BEGIN
	  IF (rdport(port) & mask) = space
	     THEN CALL parseit;
	END;
		Pascal is an older, block-structured language that has been used in some instruments.

Some instruments have been built with special purpose programming languages (or standard languages that were modified or extended). There are development speed benefits to having a language tuned to a specific application, but they come at the expense of training, tool maintenance and skilled developer availability.

10.7.4 High-level language compilers

The high-level language source code is normally run through a *compiler*. The compiler, like the assembler, is a program that runs on a development computer. It translates the high-level language input into object code files – files that contain some symbolic information as well as machine language instructions (or byte codes in the case of a language like Java). There is normally a program called a *linker* that links together the various object code language and system libraries used during the development process and the object code files that the developer has produced. This linked object code is then transferred to the microprocessor board. For moderate to large systems, this is often done by downloading the memory image to the computer via a communication interface (like RS-232 or LAN) or via removable media (like a floppy). This process is shown in the following figure (Figure 10.4).

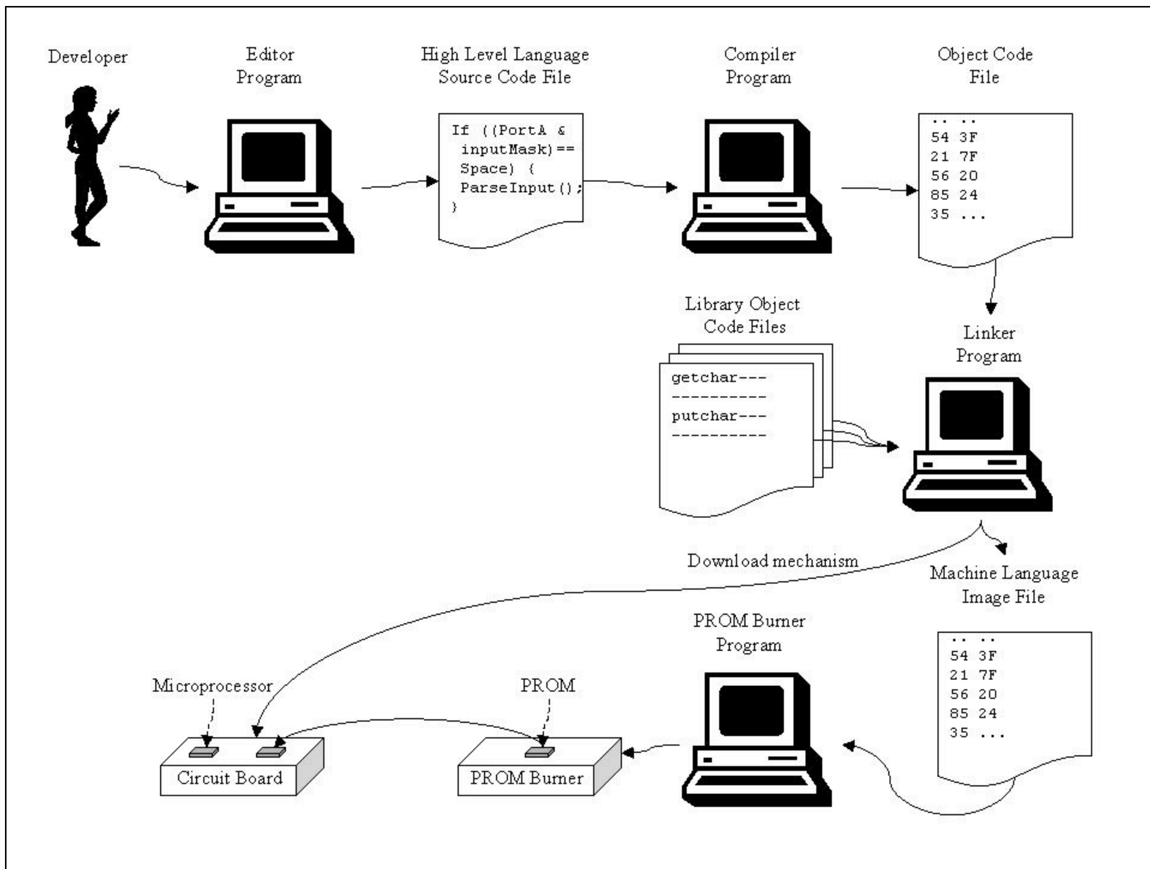

Figure 10.4: Process of high-level language to machine code

10.7.5 Operating systems software

The *Operating System* (OS) provides the software environment for the programs in the computer (embedded or otherwise). Although not all uses of embedded computers involve a true operating system, those used in most instruments do. The operating system is fundamentally the overarching program that enables the computer to do its job of executing the application program(s). Operating System services include:

- System initialization
- Resource management - memory, input and output devices
- Task management - creation, switching, priorities, communication

There are different types of operating systems and application environments described in the following table (Table 10.9):

Table 10.9: Embedded computer operating systems and application environments

OS / Environment	Description	Examples
Bootstrap loader	This is not truly an operating system, but a simple program that loads another program to run. Most often, this is called a *boot ROM* and is always in some form of nonvolatile memory (ROM or Flash). They tend to be relatively small.	Generally homegrown.

Simple executive	This is a very simple type of operating system. They tend to be simple software loops that support limited multi-tasking and scheduling. They are normally used in very simple cases.	Generally homegrown.
Real Time OS (RTOSs)	This is a special type of operating system with specific extensions designed to support high performance and real time characteristics (described in the next section 10.7.6 Real Time). This is a very common OS used in instruments. RTOSs tend to be modular and allow for relatively easy scaling.	pSOS+, VxWorks, Lynx, …
Device OS	This is a commercial operating system designed for appliances or devices, but not hard-core real time characteristics. They tend to be moderate in size, are ROM-able and do not require a hard drive.	Windows CE, JavaOS, …
Desktop OS	This is a full commercial operating system designed for PCs. They tend to be larger in size and normally require some sort of hard drive (and are not normally ROM-able). They do not have hard real-time capabilities. Real time is sometimes achieved with add-on processors or underlying software extensions.	MSDOS, DRDOS, Windows 95/98/xx, Windows NT, MacOS
Server or Multi-user OS	These are higher-end operating systems designed for multiple users. These are occasionally used in instruments – primarily in multiple-processor designs.	Unix and it variants (HP-UX, Linux, SunOS, …), Windows NT

A big part of the value and benefit of operating systems is that they support many things going on at the same time. The many different things are called processes or tasks or threads. There is not general agreement between operating system vendors about what each of these terms means. In general, the terms *process* and *task* both refer to a program and its associated state information during execution within an operating system. A *thread* is a part of program that can be run independently of the rest of the program. (Programs in an operating system can be single-threaded or multiple-threaded.) Threads tend to be light-weight and processes tend to be more complete and costly in terms of the amount of operating system overhead. The basic concept behind all of these processes, tasks and threads is that there are many things going on simultaneously. Rather than have multiple processors or very complex code, the operating system allows for multiple processes to request execution, get slices of computer processing time, and request and free system resources (like memory and hardware devices). The management of system resources is important because it manages competing access to resources. In a normal single-processor embedded system, it is important to remember that the tasks only execute one at a time (because there is a single processor). They appear to be simultaneous because of the speed of execution and the slices of computer execution given to these various processes, tasks or threads. Typical processes, tasks or threads are shown in the following table (Table 10.10).

Table 10.10: Typical embedded computer tasks

Measurement	• Calibration and correction • Hardware setup
Calculation	• Measurement algorithms • Numerical manipulation
User Interface	• Keyboard • Knobs and switches • Display (LED, LCD, CRT, Video)
External Interface	• Local peripherals • Computer interface (RS 232, IEEE 488, Parallel, …) • Network interface (LAN, …)

10.7.6 Real Time

A key aspect that an embedded computer designer/developer has to address is the required level of real time performance for the embedded computer. In the context of operating systems, the term *real time* is used to specify the ability of an operating system to respond to an external event within a specified time (*event latency*). A *real time operating system* (RTOS) is an operating system that exhibits this specified and/or guaranteed response time. This real time performance is often critical in an instrument where the embedded computer is an integral part of the measurement process and it has to deal with the flow of data in an uninterrupted fashion. Most general-purpose operating systems are not real time.

Since PC operating systems need, in general, to be responsive to the user and system events, terminology about real time has become somewhat imprecise. This has brought about some additional concepts and terms. In an operating system, there is a time window bounded by the best and worst response times. *Hard real time* is when the event response always occurs within the time window independent of other operating system activity. Note that hard real time is not necessarily fast. It could be microseconds, milliseconds or days – it is just characterized and always met. *Soft real time* is when the event response is, on average, normally in the time window. So, the system designer needs to determine if there is a need for the real time performance and whether it is hard real time or soft real time. Often, an instrument designer will use separate processors (especially DSPs) to encapsulate the real time needs and therefore simplify the overall system design.

10.8 User Interfaces

Originally, instruments used only *direct controls* - where the controls are connected directly to the analog and digital circuits. As embedded computers became common, instruments used menu or keypad driven systems (i.e. where the user input was read by the computer, which then modified the circuit operation). Today, things have progressed to the use of *Graphical User Interfaces* (GUIs). Although some instruments are intended for automated use or are 'faceless' (i.e. have no user interface), most need some way for the user to interact with the measurement or instrument. The following tables (Table 10.11 and Table 10.12) show examples of the instrument input and output devices.

Table 10.11: Typical instrument embedded computer input devices

Input device	Description
• Buttons / switches	A very common input device is a simple switch connected to I/O ports on the microprocessor. Sometimes the switch may have some pre-conditioning to remove the 'bounce' associated with key closure (which could cause spurious signals).
• RPGs	*Rotary Pulse Generators* (RPGs) are essentially digital knobs. They generate signals that are proportional to the rate, amount and direction of rotation. These are frequently used in a soft-control application in place of classical potentiometers. Some instruments have been designed with a single multi-purpose RPG whose function is set by the current instrument mode.
• Keyboard	Most instruments do not have full alphanumeric keyboards – in part due to size and in part due to the fact they are not needed. They are used in some applications like telecommunications. However, numeric keypads are very common. In most cases, the keyboard has instrument application specific keys (e.g. ATTENUATION). Generally, there is a special keyboard controller chip that takes care

	of scanning and processing key presses. Usually, the key-codes are mapped into a normal ASCII key-code for use by the microprocessor.
• Mouse / pointing device	In many instruments, there is often some form of pointing device. A mouse is useful, even necessary with a GUI, but it can be a problem given the physical environment that some instruments are located in. Often there are space or reliability problems with mice. In these cases, track-balls, touch screens, touch pads and the like are effective replacements.

Table 10.12: Typical instrument embedded computer output devices

Output device	Description
• LEDs	*Light Emitting Diodes* (LEDs) are easily connected to the microprocessor's I/O ports.
• Multi-line displays	Multiple line displays are very common in simpler instruments with moderate user interface needs. They are often implemented with LEDs, *Liquid Crystal Displays* (LCDs), and sometimes with plasma display technology. Some are 7 segment displays (like calculators) allowing for numbers and a few characters. Some are 16 segment devices that display for alphanumeric characters. There are also dot-array style displays from simple 5x7 to full addressable pixels (*picture elements*) that allow for simple graphics. Except for some of the simple 7 segment displays, they generally have built-in control circuits so they can be directly connected to one of the microprocessor's I/O ports. This controller is very helpful, from a software point of view, because it takes over the task of scanning the display.
• Video displays	Some instruments use *Cathode Ray Tubes* (CRTs) to display information. Most current displays are computer-style displays. These can be LCD or CRT technology. LCDs have advantages in weight, power and size and have improved in visual quality.
• Speaker	Audio output is occasionally used in instruments, mostly for error or feedback purposes. Audio output has the disadvantage that in some instrument environments, it is difficult to hear.

All of these user interface devices can be mixed with direct control devices (like meters, switches, potentiometers/knobs, et cetera). There are a variety of design challenges in developing effective user interfaces in instruments. However, a full discussion of instrument user interfaces is beyond the scope of this chapter. For additional information refer to Chapter ???.

10.9 External Interfaces

Most instruments include external interfaces to a peripheral, another instrument device or to an external computer. An interface to a peripheral allows the instrument to use the external peripheral – normally printing or plotting the measurement data. An interface to another device allows the instrument to communicate with or control another measurement device. The computer interface provides a communication channel between the embedded computer and the external computer. This allows the user to:

- Log (i.e. capture and store) measurement results
- Create complex automatic tests

- Combine instruments into systems
- Coordinate stimulus and response between instruments

The external computer accomplishes these tasks by transferring data, control, set-up, and/or timing information to the embedded computer. At the core of each of these interfaces is a mechanism to send and receive a stream of data bytes.

10.9.1 Hardware interface characteristics

Each interface that has been used and developed has a variety of interesting characteristics, quirks and tradeoffs. However, there are some common characteristics to understand and consider considering external interfaces:

- **Parallel or serial**: How is information sent (a bit at a time or a byte at a time)?
- **Point to point or bus/network**: How many devices are connected via the external interface?
- **Synchronous or asynchronous**: How is the data clocked between the devices?
- **Speed**: What is the data rate?

The following table (Table 10.13) describes these key characteristics.

Table 10.13: External interface characteristics

Characteristic	Description
• Parallel or serial	Probably the most fundamental characteristic of hardware interfaces is whether they send the data stream one bit at a time (*serial*) or all together (*parallel*). Most interfaces are serial. The advantage of serial is it limits the number of wires - down to a minimum of 2 lines (data and ground). However, even with a serial interface, there are often additional lines used (transmitted data, received data, ground, power, request to send, clear to send, et cetera). Parallel interfaces are normally 8 bit or 16 bit. There are older instruments that had custom *Binary Coded Decimal* (BCD) interfaces - these usually had six sets of 4 bit BCD data lines. Parallel interfaces use additional lines for handshaking – explicit indications of data ready from the sender and ready for data from the receiver.
• Point to point or bus/network	The other key characteristic of hardware interfaces is whether the interface is point to point (between the external computer and device/instrument) or some multiple device interface. The point to point interfaces can be very simple. The multiple device interfaces are sometimes implemented as a bus or a network. These bus or network interfaces require some form of device addressing. Many also need some form of conflict resolution to determine who currently has the right to send data. Network style interfaces are usually packet oriented – they send addressed blocks of data.
• Synchronous or asynchronous	Synchronous refers to interfaces that use a clock (either a separate line or one that can be reconstituted from the signal) to determine when the data is to be read. Asynchronous refers to interfaces that do not have a separate clock signal, so they depend on the data coming at a relatively consistent rate. Synchronous interfaces are generally faster than asynchronous interfaces.
• Speed	The raw speed of the interface is a key characteristic. This is normally talked about in *baud* (signaling elements per second) or *bps* (bits per second) for serial interfaces and Mbits (millions of bits per second) for network interfaces. Asynchronous serial interfaces

	tend to be in the 110 bps to 56 Kbps range. Network interfaces tend to be in the 1 Mbit to 100 Mbit range.

10.9.2 Hardware interface standards

The common hardware interfaces used with instruments are:

Table 10.14: Common instrument external interfaces

Interface	Description	Serial/Parallel	Point/Network
IEEE 488.1, *Hewlett-Packard Instrument Bus* (HP-IB), *General Purpose Instrument Bus* (GPIB)	The IEEE 488 interface has been one of the primary instrument interfaces. It allows (electrically) up to 15 devices up to 2 meters apart (each). It can achieve 1 Mbyte/sec transfer rates.	Parallel • 8 data bits • 3 wire handshake (DAV, NRFD, NDAC) • 5 additional lines (ATN, IFC, EOI, SRQ, REN)	Bus • up to 2 m apart • 31 addresses • talker • listener(s) • controller
RS-232C, EIA-232-D, RS-422A, RS-449, RS-530, CCITT V.24, IEEE 1174, MIL-188C	*RS-232C* (Recommended Standard 232C) is the venerable serial interface from the Electronic Industries Association (EIA). Almost all computers have some form of RS-232 interface. An ongoing challenge with RS-232 is configuration – which is not defined. This configuration normally includes data rate, number of stop bits, connector size (9pin, 15 pin), connector style (male or female), hardware handshake settings, which pin has the transmit data and which pin has the receive data (pin 2 and 3 problems). The other interfaces on the left are variations of the RS-232 standard with variations for data rates, configuration and connectors. Data rates for RS-232 are up to 56 Kbit/second. The variants can get above 10Mbit/second.	Serial • 2 data lines: transmitted data: TD received data: RD • 6 optional additional lines (RTS, CTS, DTR, DSR, DCD, RI)	Point to point • up to ~75 feet apart
Centronics, EPP, ECP, IEEE 1284	The Centronics interface is a standard output (one-direction) parallel interface used for connecting computers and instruments to printers or other peripherals. It has also been used as an instrument interface (connecting the computer to the instrument). The original interface has been extended to the ECP (the Extended Capabilities Port) and EPP (the Enhanced Parallel Port). IEEE Std.1284-1994 (Standard Signaling	Parallel • 8 data bits • 3 wire handshake (Strobe, Busy, Acknowledge) • 6 additional lines (Paper out, Error, Auto Feed, Select In, Select Out, Initialize)	Point to point • up to 10 m apart

	Method for a Bi-directional Parallel Peripheral Interface for Personal Computers) defines a superset that supports Centronics, ECP, EPP and two other parallel interface modes (nibble and bi-directional). The EPP and ECP extensions are both bi-directional interfaces that are about 10 times faster. In general, data rates tend to be on the order of 50-100 Kbytes/sec but it is possible to achieve up to 2 Mbytes/sec.		
Universal Serial Bus	USB (Universal Serial Bus) is an interface between a PC and devices (targeted at consumer peripherals). The USB peripheral bus is a multi-company industry standard. USB supports a data speed of 12 megabits per second.	Serial • 1 differential data line pair: data- & data+ • Supplied voltage	Network • up to ~3-4 m apart • 127 addresses (not including bridges or extenders) • peer to peer
FireWire (IEEE 1394)	FireWire (Apple's version) or IEEE 1394 High Performance Serial Bus is another interface between a PC and devices. It provides a single plug-and-socket connection on which up to 63 devices can be attached with data transfer speeds up to 400 Mbps (and eventually more). It is intended for high-performance devices.	Serial • 2 differential data line pairs: TPA, TPA# TPB, TPB# • Supplied voltage	Network • up to 4.5 m apart • 63 addresses (not including bridges or extenders) • peer to peer
Ethernet or LAN (IEEE 802.3)	Ethernet is a local-area network (LAN) protocol and hardware specification developed in 1976 for communication between computers. It supports transfer rates of 10 Mbps. Although originally on medium to large computers, it is available on PCs and is also implemented in several instruments. 100Base-T is one of several variations of Ethernet that happens to support 100 Mbps.	Serial	Network • up to ~100-500+ m apart • 2^{48} addresses

10.9.3 Software protocol standards

The previous interfaces provide the hardware interface between devices and computers. The physical layer is necessary, but not sufficient. To actually exchange information, the devices (and/or computers) require defined ways to communicate called *protocols*. If a designer is defining and building both devices that communicate, it is possible to define special protocols (simple or complex). However, most devices need to communicate with standard peripherals and computers that have predefined protocols that need to be supported. The protocols can be very complex and layered (one protocol built on top of another). This is especially true of networked or bus devices. Some of the common protocols used in instruments include:

Table 10.15: Common instrument software protocols

Software protocol	Description
IEEE 488.2	The *IEEE 488.2, Codes, Formats, Protocols and Common Commands for Use with IEE 488.1* is a specification that defines: 39 common commands and queries for instruments, the syntax for new commands and queries, and a set of protocols for how a computer and the instrument interact in various situations. Although this is a companion to the IEEE 488.1 interface, it is independent of the actual interface, but it does depend on certain interface characteristics.
SCPI	The *Standard Commands for Programmable Instruments* (SCPI) is a specification of common syntax and commands so that similar instruments from different vendors can be sent the same commands for common operations. It also specifies how to add new commands, if not currently covered in the standard.
TCP/IP	The *Transmission Control Protocol/Internet Protocol* (TCP/IP) specification is the underlying protocol used to connect devices over network hardware interfaces. The network hardware interface can be a local area network (LAN) or a wide area network (WAN).
FTP	The *File Transfer Protocol* (FTP) specification is a protocol used to request and transfer files between devices over network hardware interfaces.
HTTP	The *HyperText Transfer Protocol* (HTTP) specification is a protocol used to request and transfer web (HyperText Markup Language - HTML) pages over network hardware interfaces.
VXI-11	The VXI-11 Plug and Play specification is a protocol for communicating with instruments that use the VXI-bus (an instrument adaptation of the VME bus), GPIB/HP-IB, or a network hardware interface.

A full discussion of interface protocols is beyond the scope of this chapter, but it is covered in Chapter ???.

10.10 Numerical issues

The hardware used to construct microprocessors is only capable of manipulating ones and zeros. From these, the microprocessors build up the ability to represent integers and real numbers (and characters for that matter). Microprocessors represent all information in a binary form.

10.10.1 Integers

Integers are directly represented as patterns of ones and zeros. Each bit corresponds to a subsequent power of two – from right to left. So "0101" is equal to 2^2+2^0 or 5. Most operations on integer data in microprocessors use *unsigned binary* or *two's complement* representation. The unsigned binary has the left-most bit representing a power of two – so in a 16 bit number, a 1 in the left most bit has the decimal value of 32768. In unsigned 16 bit binary, "1000 0000 0000 0001" is equal to $2^{15}+2^0$ or 32769. The two's complement binary has the left-most bit representing a negative number. Again in a 16-bit number, the left most bit has the value of –32768 which is then added to the rest of the number[4]. So, in two's complement 16 bit binary, "1000 0000 0000 0001" is equal to $-2^{15}+2^0$ or -32768+1 which is -32767. The range of numbers in 16 bit binary two's complement and unsigned representation is shown in the following figure (Figure 10.5):

number **two's complement**

number	unsigned
65535	1111 1111 1111 1111
65534	1111 1111 1111 1110
65533	1111 1111 1111 1101

[4] This can also be viewed that a 1 in the left most bit of a two's complement number indicates a negative number where the rest of the bits need to be flipped and 1 added to the result.

32767	0111 1111 1111 1111
32766	0111 1111 1111 1110
32765	0111 1111 1111 1101
…	…
2	0000 0000 0000 0010
1	0000 0000 0000 0001
0	0000 0000 0000 0000
-1	1111 1111 1111 1111
-2	1111 1111 1111 1110
…	…
-32766	1000 0000 0000 0010
-32767	1000 0000 0000 0001
-32768	1000 0000 0000 0000

…	…
32769	1000 0000 0000 0001
32768	1000 0000 0000 0000
32767	0111 1111 1111 1111
32766	0111 1111 1111 1110
32765	0111 1111 1111 1101
…	…
2	0000 0000 0000 0010
1	0000 0000 0000 0001
0	0000 0000 0000 0000

Figure 10.5: 16 bit two's complement and unsigned binary representation

10.10.2 Floating Point Numbers

Given the nature of the vast majority of instruments, representing and operating on real numbers is necessary. Real numbers can be fixed point (a fixed number of digits on either side of the decimal point) or floating point (with a mantissa of some number of digits and an exponent). In the design of microprocessor arithmetic and math operations, numbers have a defined number of bits or digits. The value of floating point numbers is that they can represent very large and very small numbers (near zero) in this limited number of bits or digits. An example of this is the scientific notation format of the number 6.022×10^{23}. The disadvantage of floating point numbers is that, in normal implementation, they have limited precision. In the example, the number is only specified to within 10^{20}, with an implied accuracy of $+/-5 \times 10^{19}$.

Although there are several techniques for representing floating point numbers, the most common is the format defined in IEEE 754 floating point standard. There are two formats in common use from this standard, a 32-bit and a 64-bit format. In each, a number is made up of a sign, a mantissa, and an exponent. There is a fair amount of complexity in the IEEE floating point standard. The following discussion touches on the key aspects. There are aspects of IEEE floating point representation beyond the scope of this discussion. In particular, there are a variety of values represented that indicate infinity, underflow and not a number. For more detailed information on these topics (and more), refer to the IEEE standard.

For the 32 bit format, the exponent is 8 bits, the sign is one bit, and the mantissa is 23 bits. If the exponent is non-zero, the mantissa is normalized (i.e. it is shifted to remove all the zeros so that the left most digit is a binary 1 with an implied binary radix point following the left most digit). In this case the left most digit in the mantissa is implied, so the mantissa has the effect of a 24 bit value, but since the left most digit is always a 1, it can be left off. If the exponent is zero, the un-normalized mantissa is 23 bits.

Number	bit 31 Sign	bits 30-23 Exponent	bits 22-0 Mantissa
+ Not a Number (+NaN)	0	1111 1111	Non-zero
+ infinity	0	1111 1111	0000 0000 0000 0000 0000 000
…	0	…	…
+ 3.2767×10^{4} (+32767)	0	1000 1101	1111 1111 1111 1100 0000 000
…	0	…	…
+ 0.125×10^{0} (+1/8)	0	0111 1100	0000 0000 0000 0000 0000 000
…	0	…	…

+ 0	0	0000 0000	0000 0000 0000 0000 0000 000
- 0	1	0000 0000	0000 0000 0000 0000 0000 000
…	1	…	…
- 0.125×10^0 (-1/8)	1	0111 1100	0000 0000 0000 0000 0000 000
…	1	…	…
- 3.2767×10^4 (-32767)	1	1000 1101	1111 1111 1111 1100 0000 000
…	1	…	…
- infinity	1	1111 1111	0000 0000 0000 0000 0000 000
- Not a Number (-NaN)	1	1111 1111	Non-zero

Figure 10.6: 32 bit IEEE 754 floating point binary representation

The exponent for 32767 is slightly different than expected because the exponent has a bias of 127 (127 is added to the binary exponent). The 64 bit format has an 11 bit exponent (with a bias of 1023) and a 52 bit mantissa.

10.10.3 Scaling and fixed point representations

Microprocessors deal with floating point numbers either through built in floating point hardware, a floating point coprocessor, or through software. When software is used, there are significant performance penalties in speed and accuracy – it may take thousands of instructions to perform a floating point operation. And when the microprocessor has floating point capability (directly or through a co-processor), there is added cost. So, in most instrumentation applications, integers are used whenever possible. Often, instrument developers will use integer scaling (an implied offset or multiplier) to allow for some of the benefits of floating point with integer performance.

Fixed point number representation is one example of this technique. In this case, there is an implied radix point at a programmer-defined location in the binary number. For example, a pressure sensor needs to represent pressure variations in the range of –4 to +6 Pascals in steps of $1/16^{th}$ Pascal. A good fixed point representation of this is to take a two's complement signed 8 bit binary number, and put the implied radix point in the middle. This gives a range of –8 to 7.9375 with a resolution of $1/16^{th}$.

number	Fixed point scaling
7.9375	0111 1111
7.8750	0111 1110
…	…
0.0625	0000 0001
0.0000	0000 0000
-0.0625	1111 1111
…	…
-7.9375	1000 0001
-8.0000	1000 0000

Figure 10.7: 8 bit fixed point scaling binary representation example

It is also possible to have arbitrary scaling. A specific example of this is a temperature measurement application. The thermocouple used can measure –5 degrees Celsius to 120 degrees Celsius. This is a range of 125 degrees. The thermocouple has an inherent accuracy of ½ degree Celsius. The analog hardware in the instrument measures the voltage from the thermocouple and the embedded computer in the instrument converts it into a temperature reading. It would be possible to store the temperature as a 64 bit floating point number (taking additional computing power and additional storage space). It is also possible to store the temperature as a single integer: 125 degrees x 2 for the ½ degree accuracy means there are 250

distinct values that need to be represented. The integer can be an unsigned 8 bit number (which allows for 256 values).

number	Arbitrary scaling
error	1111 1111
…	…
error	1111 1011
120.0	1111 1010
119.5	1111 1001
…	…
0.5	0000 1011
0.0	0000 1010
-0.5	0000 1001
…	…
-4.5	0000 0001
-5.0	0000 0000

Figure 10.8: 8 bit fixed point arbitrary scaling binary representation example

10.10.4 Big-endian and Little-endian

Big-endian and little-endian[5] refers to the byte order of multi-byte values in computer memory, storage and communication (so this applies to computers and to protocols). The basis for the differences is determined by where the Least Significant Bit (LSB) and the Most Significant Bit (MSB) are in the address scheme. If the LSB is located in the highest address or number byte, the computer or protocol is said to be *big-endian*. If the LSB is located in the lowest address or number byte, the computer or protocol is said to be *little-endian*.

To illustrate this, let's look at a segment of computer memory that contains the 32 bit integer value 2050.

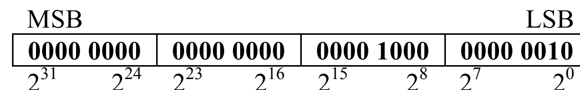

Figure 10.9: 32 bit integer for big- and little-endian examples

This value, put in a big-endian computer's memory would look like:

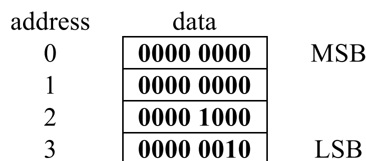

Figure 10.10: 32 bit big-endian memory layout

This value, put in a little-endian computer's memory would look like:

[5] The terms big-endian and little-endian are drawn from the Lilliputians in Gulliver's Travels who argued over which end of soft-boiled eggs should be opened.)

address	data	
0	**0000 0010**	LSB
1	**0000 1000**	
2	**0000 0000**	
3	**0000 0000**	MSB

Figure 10.11: 32 bit little-endian memory layout

Although very arbitrary, this is a serious problem. Within a computer, this is typically not an issue. However, as soon as the computer is transmitting or another external computer is accessing binary data, it is an issue because a different type of computer may be reading the data. Mainframes tend to be big-endian. PC's and the Intel 80x86 and Pentium families are little-endian. Motorola 68000 microprocessors, a popular embedded microprocessor family, are big-endian. The PowerPC is unusual because it is *bi-endian* - it supports both big-endian and little-endian styles of operation. Operating systems provide for conversions between the two orientations and the various network and communication protocols have defined ordering. Instrument users are not normally aware of this issue because developers deal with the protocols and conversions at the application software and operating system level.

10.11 Instrumentation Calibration and Correction Using Embedded Computers

Instruments normally operate in an analog world. This analog world has characteristics (like noise and non-linearity of components) that introduce inaccuracies in the instrument. Instruments generally deal with these incorrect values and try to correct them by using software in the embedded computer to provide *calibration* (adjusting for errors inside the instrument) and *correction* (adjusting for errors outside of the instrument).

10.11.1 Calibration

Calibration in the embedded computer adjusts the system to compensate for potential errors within the instrument and the instrument's probes. Embedded computer-based calibration makes hardware design much easier. In the simple case, the embedded computer can apply a calibration to correct for errors in hardware. Hardware is simpler since a linear circuit (with a reproducible result) can be corrected to the appropriate results. The calibration software in an instrument can deal with both the known inaccuracies in a family of instruments and also for the characteristics of a single instrument.

A good example of calibration in instruments is the use of Digital to Analog Converters (DACs) to calibrate analog hardware in an instrument. A DAC takes a digital value from the embedded computer and converts it to an analog voltage. This voltage is then fed into the analog circuitry of the instrument to compensate for some aspect of the instrument hardware. In a conventional oscilloscope, a probe has an adjustment to match the probe to the input impedance of the oscilloscope. A user connects the probe to a reference square-wave source and then adjusts the probe until the trace appearing on the screen is a square wave. In oscilloscopes with embedded computers, this compensation is done automatically by replacing the adjustment on the probe with a DAC and having the embedded computer adjust the DAC until it has achieved a square wave. In many cases, such an automatic adjustment is more accurate. (To fully automate this, the embedded computer also needs to switch the input between the reference input and the user's input.)

Another common example of calibration is *auto zeroing*. Many measurements can be improved by eliminating offsets. However, the offsets can be tremendously variable depending on many factors like temperature, humidity, et cetera. To achieve this, measuring devices like a voltmeter will alternate between

measuring the user's input and a short. The embedded computer can then subtract the offsets found in the zero measurement from the actual measurement, achieving a much more accurate result.

10.11.2 Linear Calibration

Linear calibrations are one of the most basic and common algorithms used by embedded computers in instrumentation. Typically, the instrument will automatically measure a calibration standard and use the result of this measurement to calibrate further measurements. The calibration standard may be internal to the instrument, in which case the operation can be completely transparent to the user. In some cases the user may be prompted to apply appropriate calibration standards as a part of the calibration procedure.

A linear calibration normally requires the user applying two known values to the input. The embedded computer makes a measurement with each input value and then calculates the coefficients to provide the calibrated output. Typically, one of the two known values will be at zero and the other at full scale. The linear calibration will be based on a formula like the following equation (Equation 10.1):

$$V_{calibrated} = \frac{V_{raw} - V_{short}}{m}$$

Equation 10.1: A sample linear calibration formula

This is a simple formula that is relatively quick to perform. However, sometimes, there are non-linear errors in the measurement system. For example, a parabola in the response curve might indicate a second-order error. Even though a linear calibration will result in a better reading in this case, a linear calibration cannot correct a non-linear error. Many instruments address this problem with non-linear calibrations. This can be done using a higher-order calibration formula like the following equation (Equation 10.2):

$$V_{calibrated} = a \times V_{raw}^{2} + b \times V_{raw} + c$$

Equation 10.2: A sample high-order calibration formula

An alternative to a polynomial is to use a *piece-wise* linear calibration. A piece-wise linear correction applies a linear correction, but the constants are varied based on the input. This allows a different correction to be applied in different regions of input. Piece-wise corrections have the advantage of not requiring as much calculation as the polynomial. Regardless of the technique used, high-order corrections require more calibration points and therefore a more involved calibration procedure for the user.

10.11.3 Correction

Correction in the embedded computer adjusts values to correct for an influence external to the instrument (e.g. in the user's test setup). For example, network analyzers will typically compensate for the effects of the connection to the *device under test* (DUT) displaying only the characteristics of the device under test. Often, the correction is performed by having the instrument make a measurement on an empty fixture and then compensating for any effects this may have on the measured result. Correction also may be used to compensate for the effects of a transducer. Typically, the sensor in a radio frequency power meter is corrected and when transducers are changed, a new correction table is loaded.

10.12 Using instruments that contain embedded computers

In the process of selecting or using an instrument with an embedded computer, there are a variety of common characteristics and challenges that arise. This section discusses some of the common aspects to consider:

- Instrument customization - what level of instrument modification or customization is needed?
- User access to the embedded computer - how much user access to the embedded computer as a general-purpose computer is needed?
- Environmental considerations - what is the instrument's physical environment?
- Longevity of instruments - how long will the instrument be in service?

10.12.1 Instrument customization

People who buy and use instruments sometimes want to modify their instruments. This is done for a variety of reasons. One of the most common reasons is **extension**: allowing the user to extend the instrument so that it can perform new instrument or measurement operations. Another very common reason is **ease of use**: customizing the instrument so that the user doesn't have to remember a difficult configuration and/or operation sequence. A less common, but still important, customization is **limitation**: this modifies the instrument by preventing access to some instrument functionality by an unauthorized user. Often this type of customization is needed for safety or security reasons - usually in manufacturing or factory floor situations.

There are several common approaches used in instruments to accommodate customization:

- **Instrument configuration** is the built-in mechanism to set the options supported by the instrument. These can include modes of operation, user language, external interfacing options, et cetera.
- **User defined keys, toolbars, and menus** are instrument user interface options that allow a user to define key sequences or new menu items customized to suit their needs.
- **Command language extensions** are instrument external interface commands that the user can define. These are normally simple command sequences - macros or batch files. A *macro* is defined as a symbol or command that represents a series of commands, actions, or keystrokes.
- **Embedded programming languages** are mechanisms built into the instrument that allow the instrument to receive and run new software (either complete applications or extensions to the current applications). In some cases, these programs are entered from the front panel. In most cases, they are provided by the manufacturer on a storage media (3.5 inch floppy, PCMCIA memory card, et cetera) or downloaded over an external interface (IEEE 488, RS-232 or LAN).

This modification set is also important to instrument manufacturers. It allows the creation of instrument **personalities**: modifying or extending instruments for custom applications or markets. Personalities are very effective for manufacturers because it allows the development of what is essentially a new product without going through the process and expense of a full release cycle.

One aspect to keep in mind about all of these customization technologies is that they do inherently change the instrument so that it is no longer a 'standard' unit. This may cause problems or have implications for service and also for use because the modified instrument may not work or act like the original, unmodified, instrument.

The user accessible embedded programming language mechanism has been a popular mechanism in instruments. Typical accessible languages are Basic, Java and C or C++. The following table (Table 10.16) discusses these languages. It also shows a simple example program in each language that reads some data from a port ten times and checks for an error condition.

Table 10.16: Common user-accessible embedded programming languages

Language	Example	Comments
Basic	10 Full = 127 20 FOR I=1 TO 10 30 MyData = Read(Port) 40 IF MyData > Full GOTO MyError 50 NEXT I	Basic has been one of the most common languages included inside instruments. It is easy to learn and use. It is almost always interpreted, which generally means there are performance limitations. Unfortunately, there are many dialects of Basic with minor variations and extensions to the language.

Java	int full = 127;
	for(int i = 1; i <= 10; i ++) {
	  int myData = read(port);
	  if (myData > full) throw(myError);
	}
		Although Java can be used in the instrument software development, it can also be used as a user accessible language. It is interpreted, so it also has performance limitations (but usually not as severe as Basic). To be useful, extensions to the language through class libraries are required. There are difficulties in that the extensions require access to the underlying instrument architecture which expose the inner workings (which are often proprietary).

C/C++	int i, myData, full;
	full = 127;
	for(i = 1; i <= 10; i ++) {
	  myData = read(port);
	  if (myData > full) myError();
	}
		Although C and C++ are very common instrument implementation languages, they can also be used in an instrument as a user accessible language. They are very good at producing customizations that are small and fast. They are not very appropriate for end users because of the difficulty of programming. They are mostly useful for manufacturers and instrument system integrators. There are also difficulties in that the extensions need access to the underlying instrument architecture which expose the inner workings (which are often proprietary).

In addition to these standard languages, some instrument designers have created special purpose languages to extend or customize the instrument. Similar to custom embedded development languages, there can be ease of use benefits to having a custom language. However, these custom languages come at the expense of user training, user documentation, and internal instrument language development and maintenance costs.

10.12.2 User access to an embedded computer

As discussed earlier, many instruments now use an embedded PC or workstation as an integral part of the system. However, the question arises: "Is the PC or workstation visible to the user." This is also related to the ability to customize or extend the system.

Manufacturers get benefits from an embedded PC because there is less software to write and it is easy to extend the system (both hardware and software). The manufacturers get these benefits even if the PC is not visible to the end user. If the PC is visible, users often like the embedded PC because it is easy to extend the system, the extensions (hardware and software) are lower cost, and they don't have to have a separate PC.

However, there are problems in having a visible embedded PC. For the manufacturer, making it visible exposes the internal architecture. This can be a problem because competitors can more easily examine their technologies. There are also problems in that users can modify and customize the system. This can translate into the user overwriting all or part of the system and application software. This is a serious

problem for the user, but is also a support problem for the manufacturer. Many instrument manufacturers that have faced this choice have chosen to keep the system closed (and not visible to the user) because of the severity of the support implications.

The user or purchaser of an instrument has a choice between an instrument that contains a visible embedded PC and an instrument that is 'just' an instrument (independent of whether it contains an embedded PC). It is worth considering how desirable access to the embedded PC is to the actual user of the instrument. The specific tasks that the user needs to perform using the embedded PC should be considered carefully. Generally, the instrument with a visible embedded PC is somewhat more expensive.

10.12.3 Environmental considerations

The embedded computer is (by definition and design) inside the instrument. Therefore, the computer has to survive the same environment as the instrument. Although many instruments are in ‘friendly’ environments, this is certainly not always true. Some of the specific factors that should be considered for the embedded computer by the instrument user and designer are discussed in the following table (Table 10.17).

Table 10.17: Environmental considerations for embedded computers

Environmental characteristic	Description
• Temperature and humidity	The embedded computer and its internal peripherals have to withstand the temperature and humidity of the instrument environment. Often, this requires a fan, given the current power used by embedded computers. Fans themselves can cause issues with failure, dust and blockage, which lead to over-heating problems.
• Shock and vibration	Many instruments need some form of mass storage device. Most of these devices are very sensitive to shock. It is often necessary to use special mounting or to use an alternative device (like flash memory).
• Operator interface	If the system is installed in a challenging environment, the human interface has to operate with those aspects. For example, the system can be located where there are high amounts of dust, dirt, industrial chemicals, industrial gases, coffee, soft drinks, untrained operators, et cetera. The human interface components that need to survive these threats include the keyboard, pointing devices, display, switches, knobs and removable media (like floppy disks).
• Power quality	Often the quality of power at the measurement site is suspect or poor. Most instrument power supplies are very carefully designed because of the analog aspects of the system. If the embedded computer has a separate power supply, it needs to be properly regulated taking the power quality into account. This is especially true of embedded PCs or workstations.
• Power fail	A separate issue from power quality is power failures. For simpler instruments, this tends to be less of a problem. However, for more and more instruments, the embedded computer makes power failures a real challenge. These more complex systems have a variety of issues: configuration memory in the middle of modification, buffers that need to be written to disk, et cetera.
• Radio Frequency Interference	*Radio Frequency Interference* (RFI) is a problem both from and to the embedded computer. Most embedded computers are operating at clock frequencies of tens and hundreds of MHz with wide buses (transmission lines). This causes high levels of generated RFI within, and potentially outside, the box. The designer needs to ensure that the embedded computer does not affect the instrument function or the

	function of other instruments nearby. Similarly, the computer is susceptible to RFI generated by the rest of the instrument and by devices outside of the instrument.

10.12.4 Instrument Longevity

Instruments tend to be built and used for very long periods of time. Unlike the consumer or personal computer markets, it is not unusual for a popular instrument to be manufactured for 15 years and in service for 20 years or more. This has implications on the embedded computer in the instrument for the users and manufacturers:

• Microprocessor	The microprocessor chip used in the instrument needs to be available for manufacture of the instrument through the life of the product. There also needs to be chip availability for maintenance.
• Language	If the languages used for developing or customizing the instrument are in the mainstream, they will probably be around later in the instrument's life. Custom and non-mainstream languages will become support challenges for the manufacturer and the user.
• Firmware / software	Software (applications, operating systems and utility software) will have defects and changes. Instruments will usually get different software over the life of a production run. Customer units will often be updated with these to enable general improvements or to fix critical defects. This can, over time, become a serious problem for a variety of reasons: • Is there enough memory for the upgrade? • Is there enough disk or flash space for the upgrade? • What software version is the instrument being upgraded from? • What is the hardware configuration of the instrument? • Where are the upgrades for an old instrument archived?
• Interfaces	Over time computer and instrumentation interfaces change. For example, BCD interfaces were previously not uncommon for instruments. Some good questions to ask are: • What performance is needed over the instrument's life? • What will be common on PCs over time? • Will the interface choice (hardware and drivers) be available on PCs over time?
• Removable media	As with interfaces, computer media changes as well. For example, 5.25" floppy disk media was a popular choice. It is worth considering what is media choice over the life of the instrument taking into account size, cost, capacity, writability, reliability, and long term viability of the media format.
• People	It is very important to adequately document the instrument design, support, manufacture, customization and use. Over the course of decades, people move on and knowledge can be forgotten or lost (or even destroyed).