4. Internal flash (in many systems the results of processing can be saved in nonvolatile memory: for example, system status periodically and images, songs, or speeches after suitable format compression).
5. Memory stick (or card): video, images, songs, or speeches and large storage in digital camera and mobile systems. Sony memory stick Micro (M2) is of size 15x12.5x1.2 mm and has a flash memory of 2 GB. It has a data transfer rate of 160 Mbps (mega bit per second) and PRO-HG 480 Mbps and 120 Mbps write [since Dec. 2006.]
6. External ROM or PROM for embedding software (in almost all systems other than microcontroller-based systems).
7. RAM memory buffers at ports.
8. Caches (in pipelined and superscalar microprocessors).

Table 1.1 details the functions assigned in embedded systems to the memories. ROM or PROM or EPROM embeds the software specific to the system.

Table 1.1 Functions assigned to the memories in a system

Memory Needed	*Functions*
ROM or EPROM or flash	Storing application programs from where the processor fetches the instruction codes. Storing codes for system booting, initializing, initial input data and strings. Codes for RTOS. Pointers (addresses) of various interrupt service routines (ISRs).
RAM (internal and external) and RAM for buffer	Storing the variables during program run and storing the stack. Storing input or output buffers, for example, for speech or image.
Memory stick	A flash memory stick is inserted in mobile computing system or digital-camera. It stores high definition video, images, songs, or speeches after a suitable format compression and stores large persistent data.
EEPROM or Flash	Storing nonvolatile results of processing.
Cache	Storing copies of instructions and data in advance from external primary memory and storing the results temporarily during processing.

A system embeds (locates) the following either in the internal flash or ROM, PROM or in an external flash or ROM or PROM of the microcontroller: boot-up program, initialization data, strings or pictogram for screen-display or initial state of the system, programs for various tasks, ISRs and operating system kernel. The system has RAMs for saving temporary data, stack and buffers that are needed during a program run. The system uses flash for storing nonvolatile results.

1.3.6 Input, Output and IO Ports, IO Buses and IO Interfaces

The system gets inputs from physical devices through the input ports. Examples are as follows:

1. A system gets inputs from the touch screen, keys in a keypad or keyboard, sensors and transducer circuits.
2. A controller circuit in a system gets inputs from the sensor and transducer circuits.
3. A receiver of signals or a network card gets the input from a communication system. [A communication system could be a fax or modem, or a broadcasting service.]
4. Ports receives inputs from a network or peripheral.

Consider the system in an Automatic Chocolate Vending Machine. It gets inputs from a port that collects the coins that a child inserts.

Consider the system in a mobile phone. A user inputs the mobile number through the buttons, directly or indirectly (through recall of the number from its memory). Keypad keys connect to the system through an input port.

A processor identifies each input port by its memory buffer addresses, called port addresses. Just as a memory location holding a byte or word is identified by an address, each input port is also identified by the address. The system gets the inputs by the read operations at the port addresses.

The system has output ports through which it sends output bytes to the real world. Examples are as follows:

1. Output may be sent to an light emitting diode (LED), liquid crystal display (LCD) or touch screen display panel. For example, a calculator or mobile phone system sends the output-numbers or an SMS message to the LCD display.
2. A system may send the output to a printer.
3. Output may be sent to a communication system or network.
4. A control system sends the outputs to alarms, actuators, furnaces or boilers.
5. A robot is sent output for its various motors.

Each output port is identified by its memory-buffer addresses (called port addresses). The system sends the output by a write operation to the port address.

There are also general-purpose ports for both the input and output (IO) operations. For example, a mobile phone system sends output as well as gets input through a wireless communication channel. A mobile computing system touch screen system sends output as well as gets input when a user touches the menu displayed or key on the screen.

Each IO port is also identified by an address to which the read and write operations both take place.

Ports can have serial or parallel communication with the system address and data buses. In serial communication a one-bit data line is used and bits are sent serially in successive time slots. Universal Asynchronous Receiver and Transmitter (UART) is a popular communication protocol for serial communication. In parallel communication, several data lines are used and bits are sent in parallel.

A system port may have to send output to multiple channels. A demultiplexer or multiplexer circuit is then used.

A demultiplexer is a digital circuit that sends digital outputs at any instance to one of the provided channels. The channel to which the output is sent is the one that is addressed by the channel address bits at the demultiplexer input. A demultiplexer takes the input and transfers it to a select channel output among the multiple output channels.

A multiplexer is a digital circuit that receives digital inputs at any instance from multiple channels, and sends data output only from a specific channel at an instance. The channel address bits are at multiplexer input. A multiplexer takes the input from one among the multiple input channels and transfers a selected channel input to the output.

A system unit (for example, memory unit or IO port or device) may have to be selected from among the multiple units in the system and activated. A decoder circuit when used as an address decoder decodes the input addresses and activates the selected output channel from among the many outputs. For example, there are 8 units of which one has to be selected. An address-select input of 3 bits is input to the decoder. Based on the input address, the output select line among the 8 activates. If the input address bit is 000, then the 0th output is active and the 0th unit activates. If the input address bit is 111, then the 7th output is active and the 7th unit activates.

Bus A system might have to be connected to a number of other devices and systems. A bus consists of a common set of lines to connect multiple devices, hardware units and systems for communication between any

two of these at any given instance. A bus communication protocol specifies how signals communicate on the bus. A bus may be a serial or parallel bus that transfers one or multiple data bits at an instance, respectively. The protocol also specifies the following: (i) ways of arbitration when several devices need to communicate through the bus; (ii) ways of polling bus requirement from each device at an instance; (iii) ways of daisy chaining the devices so that bus is granted to a device according to the device-priority in the chain.

For networking the distributed units or systems, there are different types of serial and parallel bus protocols: I^2C, CAN, USB, ISA, EISA and PCI. For wireless networking of systems there are 802.11, IrDA, Bluetooth and ZigBee protocols.

Chapter 3 will describe the ports, devices, buses and protocols in detail.

A system connects to external physical devices and systems through parallel or serial I/O ports. Demultiplexers and multiplexers facilitate communication of signals from multiple channels through a common path. A system often networks to the other devices and systems through an I/O bus: for example, I^2C, CAN, USB, ISA, EISA and PCI bus.

1.3.7 DAC Using a PWM and an ADC

DAC is a circuit that converts digital 8 or 10 or 12 bits to the analog output. The analog output is with respect to the reference voltage. When all input bits are equal to 1, then the analog output is the difference between the positive and negative reference pin voltages; when all input bits equal 0, then the analog output equals –ve reference pin voltage (usually 0 V).

Suppose a system needs to give the analog output of a control circuit for automation. The analog output may be to a power system for d.c. motor or furnace.

A pulse width modulator (PWM) with an integrator circuit is used for the DAC. A PWM unit in the microcontroller operates as follows: Pulse width is made proportional to the analog-output needed. PWM inputs are from 00000000 to 11111111 for an 8-bit DAC operation. The PWM unit outputs to an external integrator, which provides the desired analog output. From this information, the formula to obtain the analog output from the bits in a given PWM register with bits ranging from 00000000 to 11111111 is as follows: Analog output $V = K \cdot pw$, where K is constant and pw is the pulse width.

Suppose a circuit (external to the microcontroller) gives an output of 1.024 V when the pulse width is 50% of the total pulse time period, and 2.047 V when the width is 100%. When the width is made 25%, by reducing by half the value in the PWM output control-register, the integrator output will become 0.512 V. The constant K depends on integrator amplifier gain.

Assume that the integrator operates with a dual (plus–minus) supply. The PWM unit in the microcontroller operates by another method, which is as follows. Assume that when an integrator circuit gives an output of 1.023 V, the pulse width is 100% of the total pulse time period and of –1.024 V when the width is 0%. When the width is made 25% by reducing by half the value in an output control register, the integrator output will be 0.512 V; at 50% the output will be 0.0 V. From this information, the formula to obtain the analog output from the bits in a given PWM register ranging from 00000000 to 11111111 in both situations is as follows: Analog output $V = 0.01 . K' . (pw - 50)$, where K′ is constant and pw is pulse width time in percentage with respect to pulse time period. K′ depends on integrator amplifier gain.

Analog to Digital Converter ADC is a circuit that converts the analog input to digital 4, 8, 10 or 12 bits. The analog input is applied between the positive and negative pins and is converted with respect to the reference voltage. When input is equal to difference of reference positive and negative voltages, then all output bits equal 1; when equals negative reference voltage (usually 0 V), then all output bits equal 0.

The ADC in the system microcontroller can be used in many applications such as data acquisition systems (DAS), digital cameras, analog control systems and voice digitizing systems. Suppose a system gets the analog inputs from sensors of temperature, pressure, heart-beats and other sources in a DAS. Suppose a system gets the analog inputs from a digital camera. It has CCD (Charge Couple Device) which has tiny pixels that charge up on exposure to light. The charging of each pixel depends upon the light intensity at that point in the image. The analog inputs to the system generate from each pixel. Each pixel's analog input has to be converted into bits to enable processing in the next stage.

Suppose a system needs to read an analog input from a sensor or transducer circuit. If converted to bits by the ADC unit in the system, then these bits, after processing, can also give an output. This provides a control for automation by a combined use of ADC and DAC features.

The converted bits can be given to the port meant for digital display. The bits may be transferred to a memory address, a serial port or a parallel port.

A processor may process the converted bits and generate a Pulse Code Modulated (PCM) output. PCM signals are used to digitize voice into a digital format.

Important points about the ADC are as follows.

1. Either a single or dual analog reference voltage-source is required in the ADC. It sets either the analog input's upper limit or the lower and upper limits both. For a single reference source, the lower limit is set to 0 V (ground potential). When the analog input equals the lower limit, the ADC generates all bits as 0s, and when it equals the upper limit it generates all bits as 1s. [As an example, suppose in an ADC the upper limit or reference voltage is set to 2.255 V. Let the lower limit reference voltage be 0.255 V. The difference in the limits is 2 V. Therefore, the resolution will be 2/256 V. If the 8-bit ADC analog-input is 0.255 V, the converted 8 bits will be 00000000. When the input is 0.255 V + 1.000 V = 1.255 V, the bits will be 10000000. When the analog input is 0.255 V + 0.50 V, the converted bits will be 01000000. [From this information, finding a formula to obtain converted bits for a given analog input = v volt is as follows: Binary number n bits after conversion in an n-bit ADC corresponds to decimal number N. Then $N = v. (V_{ref+} - V_{ref-})/2^n$. Here, V_{ref+} is the reference voltage that gives all the bits that are equal to 1 and V_{ref-} is the reference voltage that gives all the bits that are equal to 0.]
2. An ADC may be of 8, 10, 12, or 16 bits depending upon the resolution needed for conversion.
3. The start of the conversion (STC) signal or input initiates the conversion to 8 bits. In a system, an instruction or a timer signals the STC.
4. There is an end of conversion (EOC) signal. A flag in a register is set to indicate the end of conversion and the ADC generates an interrupt; the ISR reads the ADC bits and saves them in the memory buffer.
5. There is a conversion time limit in which the conversion is definite.
6. A Sample and Hold (S/H) unit is used to sample the input for a fixed time and hold till conversion is over.

An ADC unit can be repeatedly used after the intervals equal to the conversion time. Therefore, one can digitizes the DAS sensor signals, CCD signals, voice, music or video signals, or heart beat sensor signals in different systems. An ADC unit in an embedded system microcontroller may have multichannels. It can then take the inputs in succession from the various pins interconnected to different analog sources.

For automatic control and signal processing applications, a system provides necessary interfacing circuit and software for the Digital to Analog Conversion (DAC) unit and Analog to Digital Conversion (ADC) unit. A DAC operation is done with the help of a combination of a PWM unit in the microcontroller and an external integrator chip. ADC operations are required for data acquisition, image processing, voice processing, video processing, instrumentation and automatic control systems.

1.3.8 LCD, LED and Touchscreen Displays

A system requires an interfacing circuit and software to display the status or message for a line, for multiline displays, or for flashing displays. An LCD screen may show up a multiline display of characters or also show a small graph or icon (called a pictogram). A recent innovation in the mobile phone system turns the screen blue to indicate an incoming call. Third generation system phones have both image and graphic displays. An LCD needs little power. A supply or battery (a solar panel in the calculator) powers it. The LCD is a diode that absorbs or emits light and 3 to 4 V and 50 or 60 Hz voltage-pulses with currents less than ~50 μA are required. The pulses are applied with the same polarity on the crystal front and back plane for no light, and with opposite polarity for light. Here, polarity means logic '1' or '0'. A display-controller is often used in case of matrix displays.

To indicate the ON status of the system, there may be an LED that glows. A flashing LED may indicate that a specific task is under completion or is running or in wait status. The LED is a diode that emits yellow, green, red or infrared light in a remote controller on application of a forward voltage of between 1.6–2 V. It needs current up to 12 mA above 5 mA (less in flashing display mode). It is much brighter than the LCD, making it suitable for flashing displays and for displays limited to a few digits.

A touchscreen is an input as well as an output device, which can be used to enter a command, a chosen menu or to give a reply. The information is input by physically touching at a screen position using a finger or a stylus. A stylus is thin pencil-shaped object. It is held between the fingers and used just as a pen. The screen displays the choices or commands, menus, dialog boxes and icons. The display-screen display is similar to a computer video display unit screen. Newer touch screen senses the fingers even from proximity, for example, in Apple iPhone.

Sections 3.3.4 and 3.3.5 describe the LCD and touchscreen devices and their connections to the system.

The system may need the necessary interfacing circuit and software for the output to the LCD display controller and the LED interfacing ports or for the I/Os with the touchscreen.

1.3.9 Keypad/Keyboard

The keypad or keyboard is an important device for getting user inputs. The system provides the necessary interfacing and key-debouncing circuit as well as the software for the system to receive input from a set of keys, from a keyboard, keypad or virtual keypad. A touchscreen provides for a virtual keypad in a mobile computing system. A virtual keypad is a keypad displayed on the touch screen where the user can enter the keys using a stylus or finger.

A keypad has upto a maximum of 32 keys. A keyboard may have 104 keys or more. The keypad or keyboard may interface serially or parallelly to the processor directly through ports or through a controller. Mobile phones may have a T9 keypad. A T9 keypad has 16 keys and four up-down right-left menu keys. Using 0 to 9 keys text messages, such as SMS messages, are generated.

For inputs, a keypad or board may interface to a system. The system provides necessary interfacing circuit and software to receive inputs directly from the keys or through a controller.

1.3.10 Pulse Dialer, Modem and Transceiver

For user connectivity through the telephone line, wireless or a network, a system provides the necessary interfacing and circuits. It also provides the software for pulse dialing through the telephone line, for modem

interconnection for fax, for Internet packets routing and for transmitting and connecting to a wireless cellular system or personal area wireless network. *A transceiver is a circuit that can transmit as well as receive byte streams.*

In communication system, a pulse dialer, modem or transceiver is used. A system provides the necessary interfacing circuit and software for dialing and for the modem and transceiver, directly or through a controller.

1.3.11 Interrupt Handler

A timing device sends a time-out interrupt when a preset time elapses or sends a compare interrupt when the present-time equals the preset time. Assume that data have to be transferred from a keyboard to a printer. A port peripheral generates an interrupt on receiving the input data or when the transmitting buffer becomes empty. Each action generates an interrupt. A system may possess a number of devices and the system processor has to control and handle the requirements of each device by running an appropriate ISR (interrupt service routine) for each. *An interrupts-handling mechanism must exist in each system to handle interrupts from various processes and for handling multiple interrupts simultaneously pending for service.* Chapter 4 describes in detail the interrupts, ISRs, and their handling mechanisms in a system. Important points regarding the interrupts and their handling by the program are as follows.

1. There can be a number of interrupt sources and groups of interrupt sources in a processor. [Section 4.3] An interrupt may be a hardware signal that indicates the occurrence of an event. [For example, a real-time clock continuously updates a value at a specified memory address; the transition of that value is an event that causes an interrupt.] An interrupt may also occur through timers, through an interrupting instruction of the processor program or through an error during processing. The error may arise due to an illegal op-code fetch, a division by zero result or an overflow or underflow during an ALU operation. An interrupt can also arise through a software timer. A software interrupt may arise in an exceptional condition that may have developed while running a program.
2. The system may prioritize sources and service them accordingly. [Section 4.5.]
3. Certain sources are not maskable and cannot be disabled. Some are assigned the highest priority during processing.
4. The processor's current program has to divert to a service routine to complete that task on the occurrence of the interrupt. For example, if a key is pressed, then an ISR reads the key and stores the key value in the processor memory address. If a sequence of keys is pressed, for instance in a mobile phone, then an ISR reads the keys and also calls a task to dial the mobile number.
5. There is a programmable unit on-chip for the interrupt handling mechanism in a microcontroller.
6. The operating system is expected to control the handling of interrupts and running of routines for the interrupts in a particular application. The system always gives priority to the ISRs over the tasks of an application.

A system provides an interrupt handling mechanism for executing the ISRs in case of the interrupts from physical devices, systems, software instructions and software exceptions.

1.4 EMBEDDED SOFTWARE IN A SYSTEM

The software is like the brain of the embedded system.

1.4.1 Final Machine Implementable Software for a System

An embedded system processor executes software that is specific to a given application of that system. The instruction codes and data in the final phase are placed in the ROM or flash memory for all the tasks that are executed when the system runs. The software is also called ROM image. Why? Just as an image is a unique sequence and arrangement of pixels, embedded software is also a unique placement and arrangement of bytes for instructions and data.

Each code or datum is available only in the bits and bytes format. The system requires bytes at each ROM address, according to the tasks being executed. A *machine implementable software file is therefore like a table having in each rows the address and bytes. The bytes are saved at each address of the system memory.* The table has to be readied as a ROM image for the targeted hardware. Figure 1.5 shows the ROM image in a system memory. The image consists of the boot up program, stacks address pointers, program counter address pointers, application programs, ISRs, RTOS, input data and vector addresses.

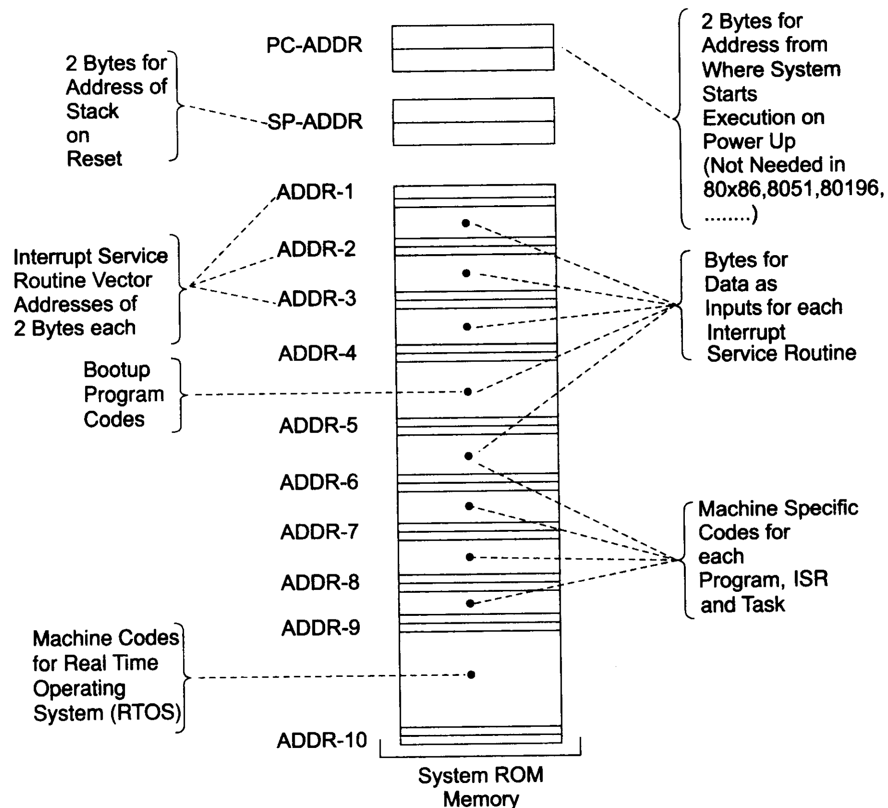

Fig. 1.5 System ROM memory embedding the software, RTOS, data and vector addresses

Final stage software is also called the ROM image. The final machine implementable software for a product embeds in the once programmable flash or ROM (or PROM) as an image in a frame. Bytes at each address must be defined to create the ROM image. By changing this image, the same hardware platform will work differently and can be used for entirely different applications or for new upgrades of the same system.

1.4.2 Coding of Software in Machine Codes

During coding in this format, the programmer defines the addresses and the corresponding bytes or bits at each address. In configuring some specific physical device or subsystem, machine code-based coding is used. For example, in a transceiver, placing certain machine code and bits can configure it to transmit at specific megabytes per second or gigabytes per second, using specific bus and networking protocols. Another example is using certain codes for configuring a control register with the processor. During a specific code-section processing, the register can be configured to enable or disable use of its internal cache. However, coding in machine implementable codes is done only in specific situations because it is time consuming and the programmer must first have to understand the processor instructions set and then memorize the instructions and their machine codes.

1.4.3 Software in Processor Specific Assembly Language

A program or a small specific part can be coded in *assembly language* using an assembler after understanding the processor and its instruction set. Assembler is software used for developing codes in *assembly*.

Assembly language coding is extremely useful for configuring physical devices like ports, a line-display interface, ADC and DAC and reading into or transmitting from a buffer. These codes are also called low-level codes for the device driver functions. [Sections 1.4.7 and 4.2.4.] They are useful to run the processor or device-specific features and provide an optimal coding solution.

Lack of knowledge of writing device driver codes or codes that utilize the processor-specific features-invoking codes in an embedded system design team can cost a lot. A vendor may charge for the APIs and also charge intellectual property fees for each system shipped out of the company.

To make all the codes in *assembly language* may, however, be very time consuming. Full coding in assembly may be done only for a few simple, small-scale systems, such as toys, automatic chocolate vending machines, robots or data acquisition systems.

Figure 1.6 shows the process of converting an *assembly language program* into machine implementable software file and then finally obtaining a ROM image file.

1. An *assembler* translates the assembly software into the machine codes using a step called *assembling*.
2. In the next step, called *linking*, a *linker* links these codes with the other codes required. Linking is necessary because of the number of codes to be linked for the final binary file. For example, there are the standard codes to program a delay task for which there is a reference in the assembly language program. The codes for the delay must link with the assembled codes. The delay code is sequential from a certain beginning address. The assembly software code is also sequential from a certain beginning address. Both the codes have to be linked at the distinct addresses as well as at the available addresses in the system. The linked file in binary for *run* on a computer is commonly known as an executable file or simply an '.exe' file. After linking, there has to be reallocation of the sequences of placing the codes before actually placing the codes in memory.

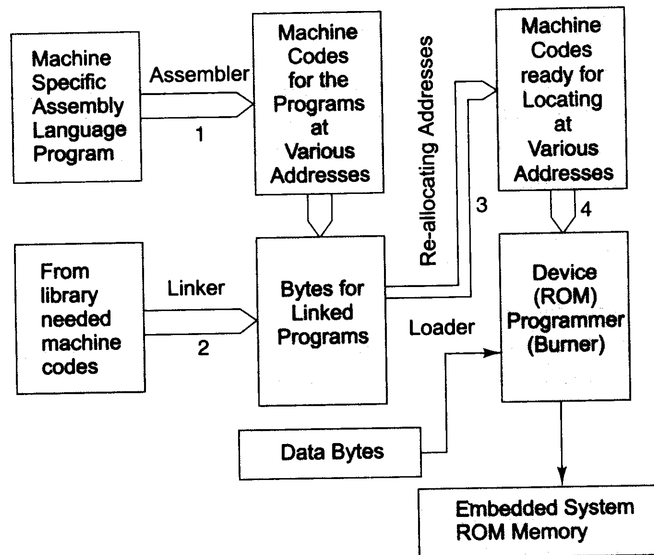

Fig. 1.6 The process of converting an assembly language program into the machine codes and finally obtaining the ROM image

3. In the next step, the *loader* program performs the task of *reallocating* the codes after finding the physical memory addresses available at a given instant. The loader is a part of the operating system and places codes into the memory after reading the '.exe' file. This step is necessary because the available memory addresses may not start from 0x0000, and binary codes have to be loaded at different addresses during the run. The loader finds the appropriate start address. In a computer, after the loader loads into a section of RAM, the program is ready to run.
4. The final step of the system design process is *locating* these codes as a ROM image. The codes are permanently placed at the addresses actually available in the ROM. In embedded systems, there is no separate program to keep track of the available addresses at different times during the run, as in a computer. In embedded systems, therefore, the next step instead of loader after linking is the use of a *locator*, which locates the IO tasks and hardware device driver codes at fixed addresses. Port and device addresses are fixed for a given system as per the interfacing circuit between the system buses and ports or devices. The *locator* program reallocates the linked file and creates a file for a permanent location of the codes in a standard format. The file format may be in the Intel Hex file format or Motorola S-record format. The designer has to define the available addresses to locate and create files to permanently locate the codes.
5. Lastly, either (i) a laboratory system, called ***device programmer,*** takes as input the ROM image file and finally *burns* the image into the PROM or flash or (ii) at a foundry, a mask is created for the ROM of the embedded system from the ROM image file. [The process of placing the codes in PROM or flash is also called burning.] The mask created from the image gives the ROM in IC chip form.

To configure some specific physical device or subsystem such as the transceiver, machine codes can be used straightaway. For physical device driver codes or codes that utilize processor-specific features-invoking codes, 'processor-specific' assembly language is used. A file is then created in three steps using an Assembler, Linker and Locator. The file has the ROM image in a standard format. A device programmer finally burns the image in the PROM or EPROM. A mask created from the image gives the ROM in IC chip form.

1.4.4 Software in High Level Language

Since the coding in *assembly language* is very time consuming in most cases, software is developed in a high-level language, 'C' or 'C++' or visual C++ or 'Java' in most cases. 'C' is usually the preferred language. The programmer needs to understand only the hardware organization when coding in high level language. As an example, consider the following problem.

Example 1.1

Add 127, 29 and 40 and print the square root.

An exemplary C language program for all the processors is as follows. (i) *# include <stdio.h>* (ii) *# include <math.h>* (iii) *void main (void) {* (iv) *int i1, i2, i3, a; float result*; (v) *i1* = 127; *i2* = 29; *i3* = 40; *a = i1 + i2 + i3; result = sqrt (a);* (vi) *printf (result);}*

The coding for square root will need many lines of code and can be done only by an expert assembly language programmer. To write the program in a high level language is very simple compared to writing it in assembly language. 'C' programs have a feature that adds the assembly instructions when using certain processor-specific features and coding for a specific section, for example, a port device driver. Figure 1.7 shows the different programming layers in a typical embedded 'C' software. These layers are as follows. (i) Processor Commands. (ii) Main Function. (iii) Interrupt Service Routine. (iv) Multiple tasks, say, 1 to N. (v) Kernel and Scheduler. (vi) Standard library functions, protocol handling and stack functions.

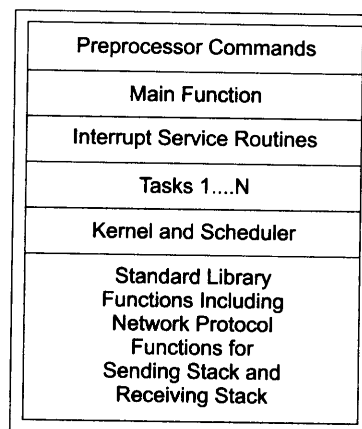

Fig. 1.7 The different program layers in the embedded software in C

Figure 1.8 shows the process of converting a C program into the ROM image file. A compiler generates the object codes. It assembles the codes according to the processor instruction set and other specifications. The C compiler for embedded systems must, as a final step of compilation, use a code-optimizer that optimizes the codes before linking. After compilation, the linker links the object codes with other needed codes. For example, the linker includes the codes for the functions *printf* and *sqrt* codes. Codes for device and driver (device control codes) management also link at this stage: for example, printer device management and driver codes. After linking, the other steps for creating a file for ROM image are the same as shown earlier in Figure 1.6.

C, C++, Java, Visual C++ are the languages used for software development. A C program has various layers: processor commands, main function, task and library functions, interrupt service routines and kernel (scheduler). The compiler generates an object file. Using a linker and locator, the file for the ROM image is created for the targeted hardware.

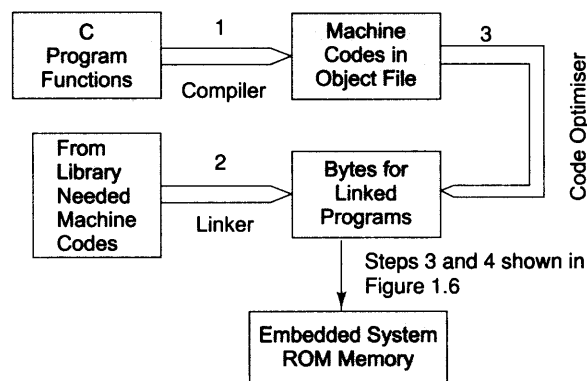

Fig. 1.8 The process of converting a C program into the file for ROM image

1.4.5 Program Models for Software Designing

The program design task is simplified if a program is modeled.

The different models that are employed during the design processes of the embedded software are as follows:

1. Sequential Program Model
2. Object Oriented Program Model
3. Control and Data flow graph or Synchronous Data Flow (SDF) Graph or Multi Thread Graph (MTG) Model
4. Finite State Machine for data path
5. Multithreaded Model for concurrent processing of processes or threads or tasks

UML (Universal Modeling language) is a modeling language for object oriented programming.

These models are explained Chapter 6.

1.4.6 Software for Concurrent Processing and Scheduling of Multiple Tasks and ISRs Using an RTOS

An embedded system program is most often designed using multiple processes or multitasks or a multithreads. [Refer to Sections 7.1 to 7.3 for definitions and understanding of the processes, threads and tasks.] The multiple tasks are processed most often by the OS not sequentially but concurrently. Concurrent processing tasks can be interrupted for running the ISRs, and a higher priority task preempts the running of lower priority tasks.

An OS provides for process, memory, devices, IOs and file system management. A file system specifies the ways in which a file is created, called, named, used, copied, saved or deleted, opened and closed. File system is the software for using the files on a disk, flash memory, memory card or memory stick.

OS software have scheduling functions for all the processes (tasks, ISRs and device drivers) in the system. Since the running of the tasks and ISRs may have real time constraints and deadlines for finishing the tasks, an RTOS is required in an embedded system. The RTOS provides the OS functions for coding the system, provides interprocess communication functions and controls the passing of messages and signals to a task.

RTOS functions are highly complex. There are a number of popular and readily available RTOSs.

Chapters 8 to 12 describes the RTOS functions and examples of applications in the embedded systems.

RTOS is used in most embedded systems and the system does concurrent processing of multiple tasks when the tasks have real time constraints and deadlines.

1.4.7 Software for Device Drivers and Device Management in an Operating System

An embedded system is designed to perform multiple functions and has to control multiple physical and virtual devices. In an embedded system, there may be number of *physical devices*. Exemplary physical devices are timers, keyboards, display, flash memory, parallel ports and network cards.

A program is also be developed using the concept of *virtual devices*. Examples of virtual devices are as follows.

1. A file (of records opened, read, written and closed, and saved as a stream of bytes or words)
2. A pipe (for sending and receiving a stream of bytes from a source to destination)
3. A socket (for sending and receiving a stream of bytes between the client and server software and between source and destination computing systems)
4. A RAM disk (for using the RAM in a way similar to files on the disk)

A file is a data structure (or virtual device) which sends the records (characters or words) to a data sink (for example, a program function) and which stores the data from the data source (for example, a program function). A file in a computer may also be stored in the hard disk and in flash memory in embedded system.

The term virtual device follows from the analogy that just as a keyboard gives an input to the processor for a *read*, a file also gives an input to the processor. The processor gives an output to a printer for a *write*. Similarly, the processor writes an output to the file.

A device for the purpose of control, handling, reading and writing actions can be taken as consisting of three components. (i) A control register or word that stores the bits that, on setting or resetting by a device driver, control device actions. (ii) A status register or word that provides the flags (bits) to show the device status to the device driver. (iii) A device mechanism that controls the device actions. There may be input and output data buffers in a device, which may be written or read by a device driver. Device driver actions are to get input into or send output from the control registers, input data buffers, output data buffers and status registers of the device.

A device driver is software for opening, connecting or binding, reading, writing and closing or controlling actions of the device. It is software written in a high level language. It controls functions for device open (configure), connect, bind, listen, read or write or close. The device driver executes after the programming of the control register (or word) of a peripheral or virtual device. The programming is called device initialisation or registration or attachment. The driver reads the status register, gets the inputs and writes the outputs. It executes on an interrupt to or from the device.

A driver controls three functions. (i) Initializing, which is activated by placing appropriate bits at the control register or word. (ii) Calling an ISR on interrupt or on setting a status flag in the status register and running (driving) the ISR (Interrupt Handler Routine). (iii) Resetting the status flag after an interrupt service. A driver may be designed for asynchronous operations (multiple use by tasks one after another) or synchronous operations (concurrent use by the tasks).

Using the functions of the OS, a device driver coding can be made such that the underlying hardware is hidden as much as possible. An API then defines the hardware separately. This makes the driver usable when the device hardware changes in a system.