EE308 MICROPROCESSORS LABORATORY

Introduction to Microprocessors Laboratory with Assembly Programming

Background Work:

- Read Section 1 of the **Jonathon Valvano's textbook - 2nd Edition**.
- Install and run **Keil uVision 5** on your personal computer and download the necessary packages.
- Download and run the installer using the following link: http://users.ece.utexas.edu/~valvano/Volume1/EE319K_Install.exe

Purpose:

The general purpose of this laboratory is to familiarize you with the software development steps using the **Keil uVision** simulator. In the following labs, we will use uVision for both simulation and debugging on the real board, but for this lab, we will just use the simulator.

You will learn how to perform **digital input/output** on parallel ports of the TM4C123. Software skills you will learn include **port initialization, logic operations, and unconditional branching**.

Do not use any conditional branches in your solution! We want you to think of the solution in terms of logical and shift operations. Logical operations include **AND, ORR** and **EOR**. Shift operations include **LSL** and **LSR**.

System Requirements:

The objective of this system is to implement an an **even parity system**. There are three bits of inputs and one bit of output. The output is in positive logic: outputing a 1 will turn on the LED, outputing a 0 will turn off the LED. Inputs are negative logic: meaning if the switch not pressed is the input is 1, if the switch is pressed the input is 0.

- PE0 is an input
- PE1 is an input
- PE2 is an input
- PE3 is the output

Even parity is an algorithm used in communication systems to detect errors during transmission. Consider the three inputs as a 3-bit data value, such that if the input switch is pressed, that data bit is 1. Your system will add one output bit, creating a 4-bit value, such that the number of zeros, considered as one 4-bit value will always be odd. The communication system (if there were one) sends the 4-bit value as a message (containing the 3-bit data plus parity), and the receiver could detect if one of the bits were to be flipped during transmission.

The specific operation of this system

- Initialize Port E to make PE0, PE1, PE2 inputs and PE3 an output
- Make the output 1 if there is an odd number of switches pressed, otherwise make output 0.
- Over and over, read the inputs, calculate the parity bit and set the parity bit at the output

The input data refers to the switch, not the input. The following table illustrates the expected behavior relative to output PE3 as a function of inputs PE0, PE1, PE2 (negative logic with respect to the switches).

PE2	PE1	PE0		PE3	number of 0's in 3 input bits
0	0	0	|	1	odd
0	0	1	|	0	even
0	1	0	|	0	even
0	1	1	|	1	odd
1	0	0	|	0	even
1	0	1	|	1	odd
1	1	0	|	1	odd
1	1	1	|	0	even

There are 8 valid output values for Port E: 0x01,0x02,0x04,0x07,0x08,0x0B,0x0D, and 0x0E. General rule, PE0, PE1, PE2, PE3 always have an odd number of 0's.

You could also consider the specification the LED output as a function of the switch input. The following table illustrates the expected behavior relative to LED output as a function of switch presses

Sw2	Sw1	Sw0		LED	number switches pressed
press	press	press	|	On	odd
press	press	not	|	Off	even
press	not	press	|	Off	even
press	not	not	|	On	odd
not	press	press	|	Off	even
not	press	not	|	On	odd
not	not	press	|	On	odd
not	not	not	|	Off	even

where press means the switch is pressed and not means the switch is not pressed.

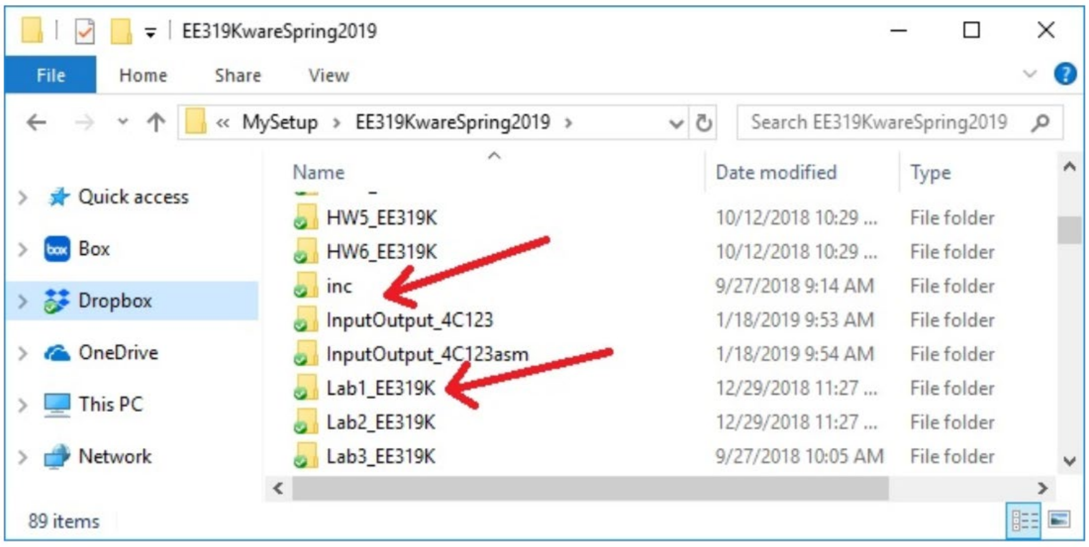
Laboratory Work:

1. Verify Keil Project for Lab1 is present and runs

To work on Lab 1, perform these tasks. Find a place on your hard drive to save all your TM4C123 software. In Figure 1 it is called **EE319KwareSpring2020**, created when you install the .exe. Download and unzip the starter configuration from:

http://users.ece.utexas.edu/~valvano/Volume1/EE319K_Install.exe

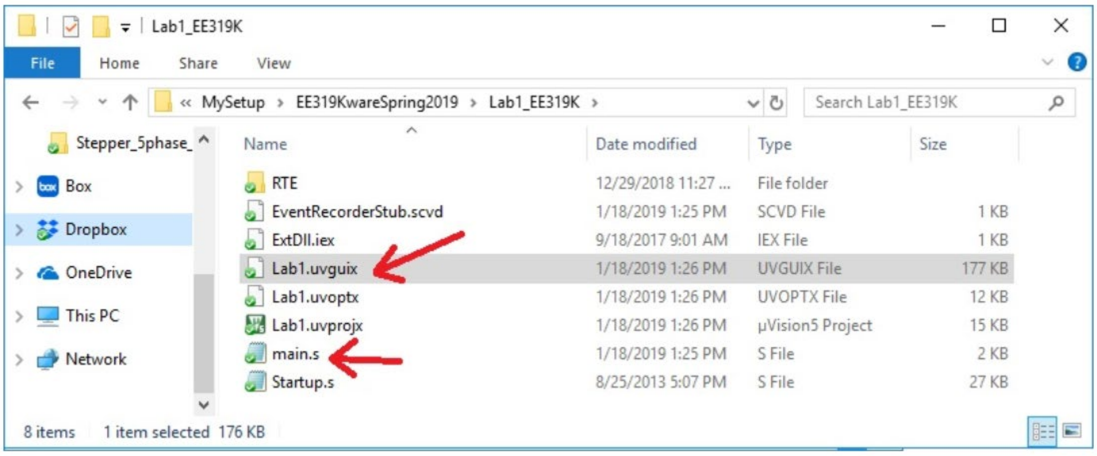
into this location. Notice the solutions to the labs will be folders that begin with "Lab".



Name	Date modified	Type
HW5_EE319K	10/12/2018 10:29 ...	File folder
HW6_EE319K	10/12/2018 10:29 ...	File folder
inc	9/27/2018 9:14 AM	File folder
InputOutput_4C123	1/18/2019 9:53 AM	File folder
InputOutput_4C123asm	1/18/2019 9:54 AM	File folder
Lab1_EE319K	12/29/2018 11:27 ...	File folder
Lab2_EE319K	12/29/2018 11:27 ...	File folder
Lab3_EE319K	9/27/2018 10:05 AM	File folder

Figure 1. Directory structure with your Lab1.

It is important for the directory structure to look like Figure 1. Notice the directory relationship between the lab folders and the **inc** (include) folder. Begin with the **Lab1_EE319K** project in the folder **EE319KwareSpring2020**. Either double click the **uvprojx** file or open the project from within uVision. Make sure you can compile it and run on the simulator. Please contact your TA if the starter project does not compile or run on the simulator. **Startup.s** contains assembly source code to define the stack, reset vector, and interrupt vectors. Some projects in this class will include this file, and you should not need to make any changes to the **Startup.s** file. **main.s** will contain your assembly source code for this lab. You will edit the **main.s** file.



Lab1_EE319K

File Home Share View

« MySetup › EE319KwareSpring2019 › Lab1_EE319K ›

Search Lab1_EE319K

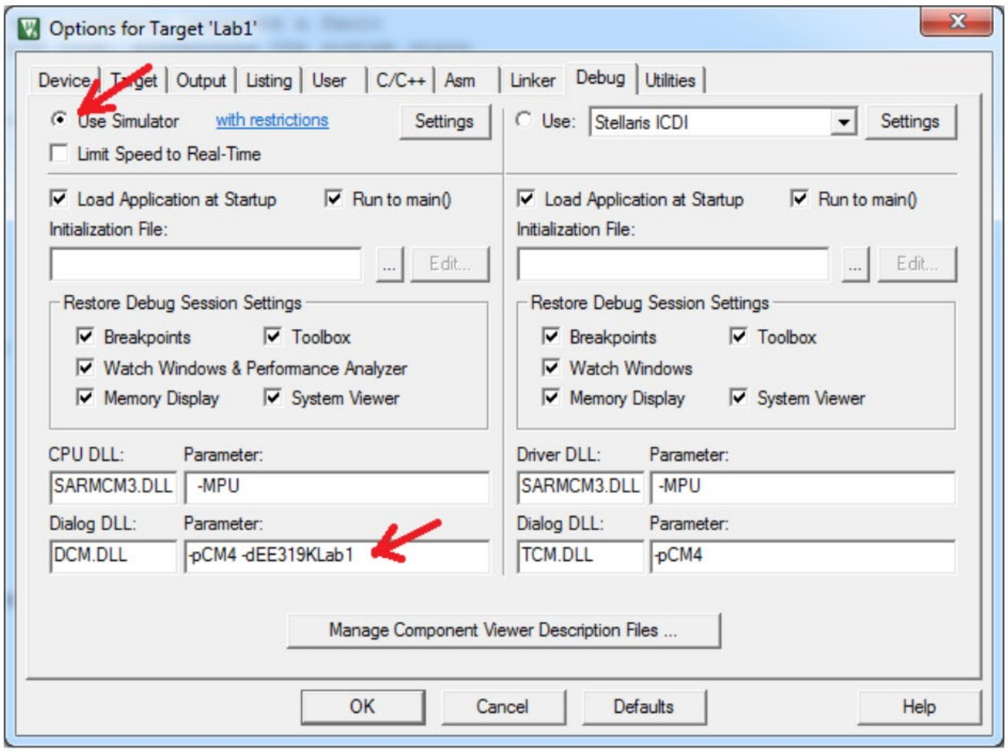
Stepper_5phase_ · Box · Dropbox · OneDrive · This PC · Network

Name	Date modified	Type	Size
RTE	12/29/2018 11:27 ...	File folder	
EventRecorderStub.scvd	1/18/2019 1:25 PM	SCVD File	1 KB
ExtDll.iex	9/18/2017 9:01 AM	IEX File	1 KB
Lab1.uvguix	1/18/2019 1:26 PM	UVGUIX File	177 KB
Lab1.uvoptx	1/18/2019 1:26 PM	UVOPTX File	12 KB
Lab1.uvprojx	1/18/2019 1:26 PM	µVision5 Project	15 KB
main.s	1/18/2019 1:25 PM	S File	2 KB
Startup.s	8/25/2013 5:07 PM	S File	27 KB

8 items 1 item selected 176 KB

Figure 2. Start Keil by opening the ***Lab1.uvprojx*** *file.*

To run the Lab 1 simulator, you must check two things. First, execute Project->Options and select the Debug tab. The debug parameter field must include **-dEE319KLab1**. Second, the **EE319KLab1.dll** file must be present in your Keil\ARM\BIN folder (the EE319K DLLs should have been put there by the installer).



Options for Target 'Lab1'

Device | Target | Output | Listing | User | C/C++ | Asm | Linker | Debug | Utilities

[✓ Use Simulator] with restrictions Settings

☐ Limit Speed to Real-Time

☑ Load Application at Startup ☑ Run to main()

Initialization File:

Restore Debug Session Settings

☑ Breakpoints ☑ Toolbox

☑ Watch Windows & Performance Analyzer

☑ Memory Display ☑ System Viewer

CPU DLL: SARMCM3.DLL Parameter: -MPU

Dialog DLL: DCM.DLL Parameter: -pCM4 -dEE319KLab1

Use: Stellaris ICDI Settings

☑ Load Application at Startup ☑ Run to main()

Initialization File:

Restore Debug Session Settings

☑ Breakpoints ☑ Toolbox

☑ Watch Windows

☑ Memory Display ☑ System Viewer

Driver DLL: SARMCM3.DLL Parameter: -MPU

Dialog DLL: TCM.DLL Parameter: -pCM4

Manage Component Viewer Description Files ...

OK Cancel Defaults Help

Figure 3. Debug the software using the simulator (DCM.DLL -pCM4 -dEE319KLab1).

2. Draw Flowchart

Write a flowchart for this program. We expect 5 to 15 symbols in the flowchart. A flowchart describes the algorithm used to solve the problem and is a visual equivalent of pseudocode. See Appendix I in this lab manual for example flowcharts and detailed explanation.

3. Write Pseudocode

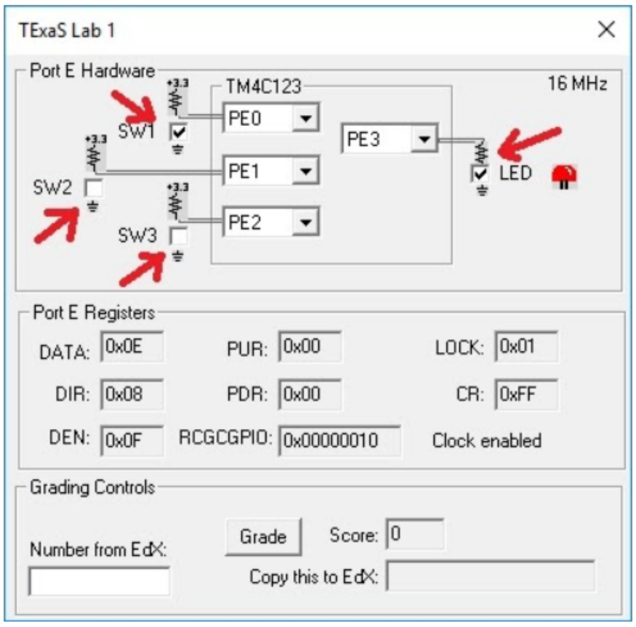
Write pseudocode for this program. We expect 5 to 10 steps in the pseudocode. You may use any syntax you wish, but the algorithm should be clear. See Example 1.17.1 in the book (Section 1.17) for an instance of what pseudocode ought to look like. Note, pseudocode ought to embody the algorithm and therefore be language blind. The same pseudocode can serve as an aid to writing the solution out in either assembly or C (or any other language).

4. Write Assembly

You will write assembly code that inputs from PE2, PE1, PE0 and outputs to PE3. The address definitions for Port E are listed below, and these are placed in the starter file main.s:

```
GPIO_PORTE_DATA_R  EQU 0x400243FC
GPIO_PORTE_DIR_R   EQU 0x40024400
GPIO_PORTE_DEN_R   EQU 0x4002451C
SYSCTL_RCGCGPIO_R  EQU 0x400FE608
```

To interact with the I/O during simulation, **make sure that View->Periodic Window Update is checked or the simulator will not update!** Then, execute the **Peripherals->TExaS Port E** command. When running the simulator, we check and uncheck bits in the I/O Port box to change input pins. We observe output pin in the window. You can also see the other registers, such as DIR DEN and RCGCGPIO.



DATA: 0x0E
PUR: 0x00
LOCK: 0x01
DIR: 0x08
PDR: 0x00
CR: 0xFF
DEN: 0x0F
RCGCGPIO: 0x00000010
Clock enabled
Score: 0

Figure 4. In simulation mode, we interact with virtual hardware. Notice there are 1 (odd) switches pressed, so the output is high.

Demonstration:

During the demonstration, **you will be asked to run your program to verify proper operation. You should be able to single-step your program and explain what your program is doing and why.** You need to know how to set and clear breakpoints. You should know how to visualize Port E input/output in the simulator.

You should bring with you the flow chart of the system and pseudocode of the algorithm on an A4 paper (one side is flowchart and the other side is pseudocode) to the lab. You flowchart and pseudocode are to be compatible with your assembly code.

IF PLAGIARISM IS DETECTED, YOU MAY GET ZERO FOR ALL THE LABS!

Appendix I:

(from section 1.8 in J. Valvano's textbook)

The remainder of this chapter will discuss the art and science of designing embedded systems from a general perspective. If you need to write a paper, you decide on a theme, and then begin with an outline. In the same manner, if you design an embedded system, you define its specification (what it does) and begin with an organizational plan. In this chapter, we will present three graphical tools to describe the organization of an embedded system: flowcharts, data flow graphs, and call graphs. You should draw all three for every system you design. In this section, we introduce the flowchart syntax that will be used throughout the book. Programs themselves are written in a linear or one-dimensional fashion. In other words, we type one line of software after another in a sequential fashion. Writing programs this way is a natural process, because the computer itself usually executes the program in a top-to-bottom sequential fashion. This one-dimensional format is fine for simple programs, but conditional branching and function calls may create complex behaviors that are not easily observed in a linear fashion. Flowcharts are one way to describe software in a two-dimensional format, specifically providing convenient mechanisms to visualize conditional branching and function calls. Flowcharts are very useful in the initial design stage of a software system to define complex algorithms. Furthermore, flowcharts can be used in the final documentation stage of a project, once the system is operational, in order to assist in its use or modification.

Figures throughout this section illustrate the syntax used to draw flowcharts (Figure 1.28). The oval shapes define entry and exit points. The main **entry point** is the starting point of the software. Each function, or subroutine, also has an entry point. The **exit point** returns the flow of control back to the place from which the function was called. When the software runs continuously, as is typically the case in an embedded system, there will be no main exit point. We use rectangles to specify **process** blocks. In a high-level flowchart, a process block might involve many operations, but in a low-level flowchart, the exact operation is defined in the rectangle. The parallelogram will be used to define an **input/output** operation. Some flowchart artists use rectangles for both processes and input/output. Since input/output operations are an important part of embedded systems, we will use the parallelogram format, which will make it easier to identify input/output in our flowcharts. The diamond-shaped objects define a branch point or **conditional** block. Inside the diamond we can define what is being tested. Each arrow out of a condition block must be labeled with the condition causing flow to go in that direction. There must be at least two arrows out of a condition block, but there could be more than two. However, the condition for each arrow must be mutually exclusive (you can't say "if I'm happy go left and if I'm tall go right" because it is unclear what you want the software to do if I'm happy and tall). Furthermore, the complete set of conditions must define all possibilities (you can't say "if temperature is less than 20 go right and if the temperature is above 40 go left" because you have not defined what to do if the temperature is between 20 and 40). The rectangle with double lines on the side specifies a call to a **predefined function**. In this book, functions, subroutines, and procedures are terms that all refer to a well-defined section of code that performs a specific operation. Functions usually return a result parameter, while procedures usually do not. Functions and procedures are terms used when describing a high-level language, while subroutines are often used when describing assembly language. When a function (or subroutine or procedure) is called, the software execution path jumps to the function, the specific operation is performed, and the execution path returns to the point immediately after the function call. Circles are used as **connectors**. A connector with an arrow pointing out of the circle defines a label or a spot in the algorithm. There should be one label connector for each number. Connectors with an arrow pointing into the circle are jumps or goto commands. When the flow reaches a goto connector, the execution path jumps to the position specified by the corresponding label connector. It is bad style to use a lot of connectors.

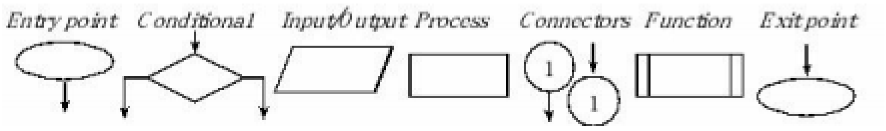


Figure 1.28. Flowchart symbols.

There are a seemingly unlimited number of tasks one can perform on a computer, and the key to developing great products is to select the correct ones. Just like hiking through the woods, we need to develop guidelines (like maps and trails) to keep us from getting lost. One of the fundamentals when developing software, regardless whether it is a microcontroller with 1000 lines of assembly code or a large computer system with billions of lines of code, is to maintain a consistent structure. One such framework is called **structured programming**. A good high-level language will force the programmer to write structured programs. Structured programs are built from three basic building blocks: the **sequence**, the **conditional**, and the **while-loop**. At the lowest level, the process block contains simple and well-defined commands. I/O functions are also low-level building blocks. Structured programming involves combining existing blocks into more complex structures, as shown in Figure 1.29.

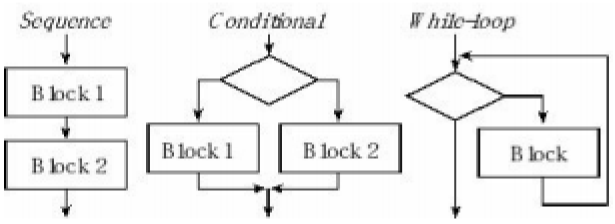


Figure 1.29. Flowchart showing the basic building blocks of structured programming.

Example 1.1: Using a flowchart describe the control algorithm that a toaster might use to cook toast. There will be a start button the user pushes to activate the machine. There is other input that measures toast temperature. The desired temperature is preprogrammed into the machine. The output is a heater, which can be on or off. The toast is automatically lowered into the oven when heat is applied and is ejected when the heat is turned off.

Solution: This example illustrates a common trait of an embedded system, that is, they perform the same set of tasks over and over forever. The program starts at **main** when power is applied, and the system behaves like a toaster until it is unplugged. Figure 1.30 shows a flowchart for one possible toaster algorithm. The system initially waits for the operator to push the start button. If the switch is not pressed, the system loops back reading and checking the switch over and over. After the start button is pressed, heat is turned on. When the toast temperature reaches the desired value, heat is turned off, and the process is repeated.

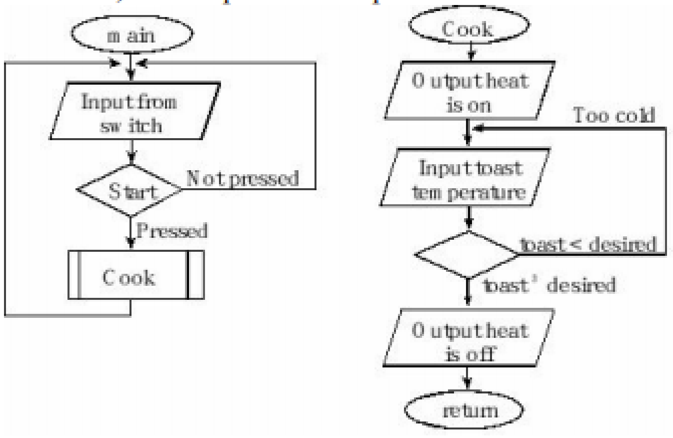


Figure 1.30. Flowchart illustrating the process of making toast.

Safety tip: When dealing with the potential for fire, you may want to add some safety features such as a time out or an independent check for temperature overflow.

Observation: The predefined functions in this chapter do not communicate any data between the calling routine and function. Data passed into a function are called input parameters, and data passed from the function back to the calling routine are called output parameters.

Observation: Notice in Figure 1.30 we defined a function Cook even though it was called from only one place. You might be tempted to think it would have been better to paste the code for the function into the one place it was called. There are many reasons it would be better to define the function as a separate software object: it will be easier to debug because there is a clear beginning and end of the function, it will make the overall system simpler to understand, and in the future we may wish to reuse this function for another purpose.

Example 1.2. The system has one input and one output. An event should be recognized when the input goes from 0 to 1 and back to 0 again. The output is initially 0, but should go 1 after four events are detected. After this point, the output should remain 1. Design a flowchart to solve this problem.

Solution: This example also illustrates the concept of a subroutine. We break a complex system into smaller components so that the system is easier to understand and easier to test. In particular, once we know how to detect an event, we will encapsulate that process into a subroutine, called **Event**. In this example, the **main** program first sets the output to zero, calls the function **Event** four times, then it sets the output to one. To detect the 0 to 1 to 0 edges in the input, it first waits for 1, and then it waits for 0 again. See Figure 1.31.

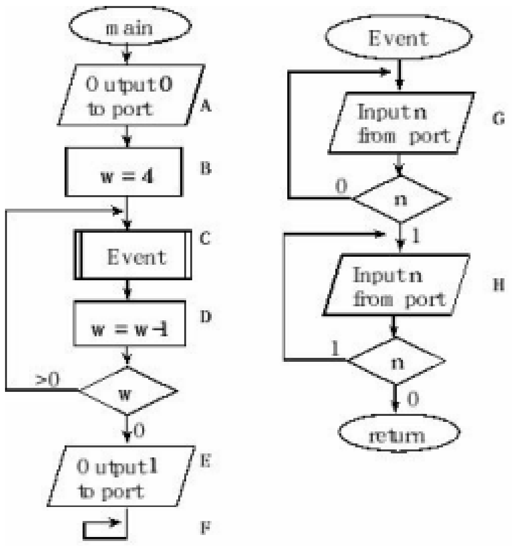
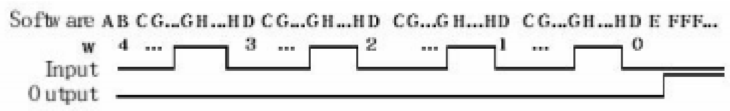


Figure 1.31. Flowchart illustrating the process waiting for four events.

The letters **A** through **H** in Figure 1.31 specify the software activities in this simple example. In this example, execution is sequential and predictable.


Software AB CG...GH...HD CG...GH...HD CG...GH...HD CG...GH...HD E FFF...
w 4 ... 3 ... 2 ... 1 ... 0
Input
Output