# UNIT-IV
# EMBEDDED SOFTWARE DEVELOPMENT TOOLS

**UNIT IV Contents at a glance:**

I. Host and target machines
II. linker/locators for embedded software
III. getting embedded software into the target system

**DEBUGGING TECHNIQUES**

IV. Testing on host machine
V. using laboratory tools
VI. an example system

## I. HOST AND TARGET MACHINES:

- **Host:**
  - A computer system on which all the programming tools run
  - Where the embedded software is developed, compiled, tested, debugged, optimized, and prior to its translation into target device.
- **Target:**
  - After writing the program, compiled, assembled and linked, it is moved to target
  - After development, the code is cross-compiled, translated – cross-assembled, linked into target processor instruction set and located into the target.

| Host System | Target Computer System |
|---|---|
| Writing, editing a program, compiling it, linking it, debugging it are done on host system | After the completion of programming work, it is moved from host system to target system. |
| It is also referred as Work Station | No other name |
| Software development is done in host system for embedded system | Developed software is shifted to customer from host |
| Compiler, linker, assembler, debugger are used | Cross compiler is also used |
| Unit testing on host system ensures software is working properly | By using cross compiler, unit testing allows to recompile code ,execute, test on target system |
| Stubs are used | Real libraries |
| Programming centric | Customer centric |

**Cross Compilers:**

- A **cross compiler that runs on host system** and produces the binary instructions that will be understood by your target microprocessor.
- *A **cross compiler** is a compiler capable of creating executable code for a platform other than the one on which the compiler is running. For example, a compiler that runs on aWindows 7 PC but generates code that runs on Android smartphone is a cross compiler.*

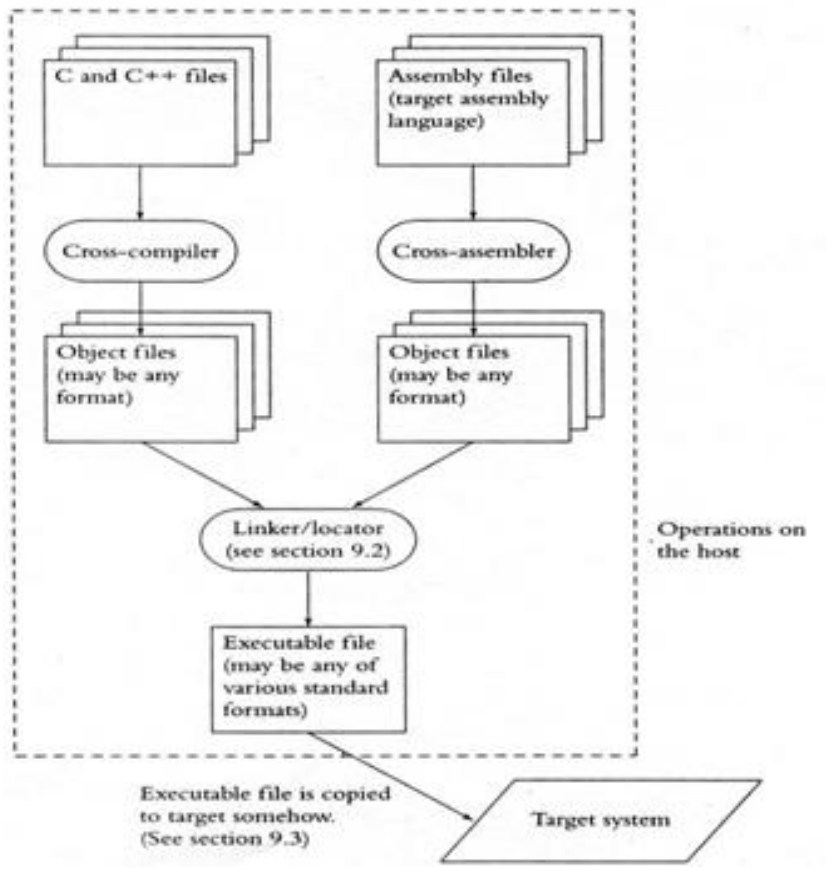
- Most desktop systems are used as hosts come with compilers, assemblers, linkers that will run on the host. These tools are called native tools.
- Suppose the native compiler on a Windows NT system is based on Intel Pentium. This compiler may possible if target microprocessor is also Intel Pentium. This is not possible if the target microprocessor is other than Intel i.e. like MOTOROLA, Zilog etc.
- A cross compiler that runs on host system and produces the binary instructions that will be understood by your target microprocessor. This cross compiler is a program which will do the above task. If we write C/C++ source code that could compile on native compiler and run on host, we could compile the same source code through cross compiler and make run it run on target also.
- That may not possible in all the cases since there is no problem with if, switch and loops statements for both compilers but there may be an error with respect to the following:
  - In Function declarations
  - The size may be different in host and target
  - Data structures may be different in two machines.
  - Ability to access 16 and 32 bit entries reside at two machines.

  Sometimes cross compiler may warn an error which may not be warned by native complier.

**Cross Assemblers and Tool Chains:**

- Cross assembling is necessary if target system cannot run an assembler itself.
- **A cross assembler is a program that runs on host produces binary instructions appropriate for the target.** The input to the cross assembler is assembly language file (.asm file) and output is binary file.
- A cross-assembler is just like any other assembler except that it runs on some CPU other than the one for which it assembles code.

**Tool chain for building embedded software shown below:**




The figure shows the process of building software for an embedded system.

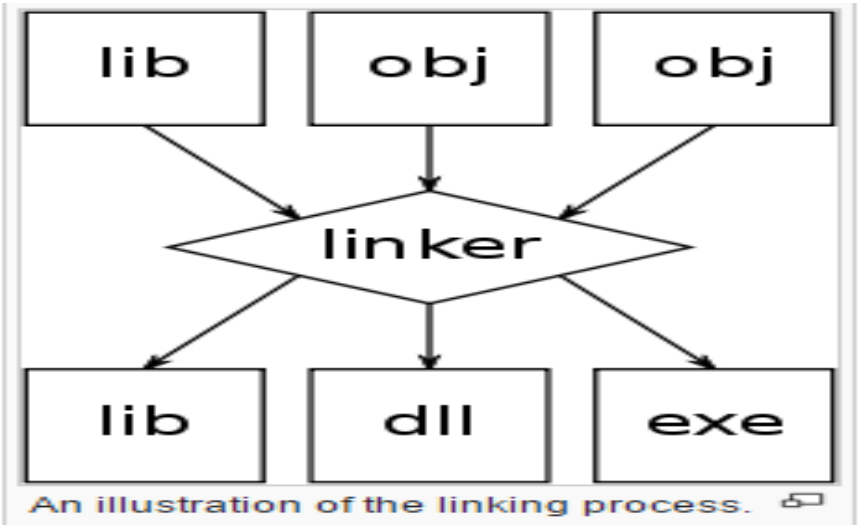
As you can see in figure the output files from each tool become the input files for the next. Because of this the tools must be compatible with each other.

A set of tools that is compatible in this way is called tool chain. Tool chains that run on various hosts and builds programs for various targets.

**II. LINKER/LOCATORS FOR EMBEDDED SOFTWARE:**

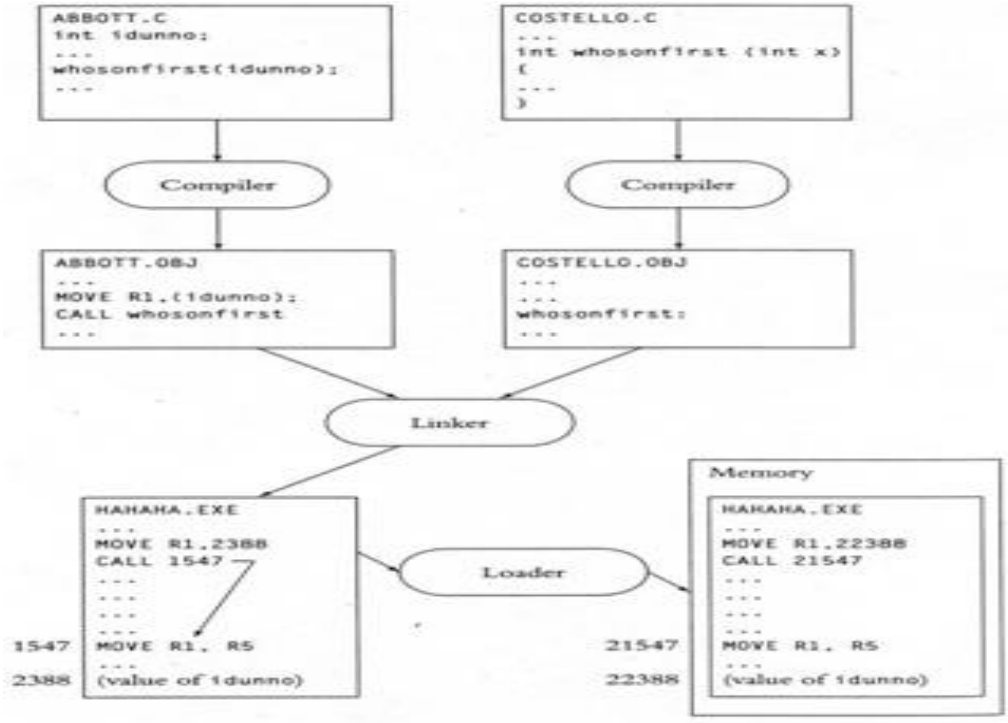
- **Linker:**
  - a linker or link editor is a computer program that takes one or more object files generated by a compiler and combines them into a single executable file, library file, or another object file.
- **Locator:**
  - *locate* embedded binary code into target processors
  - produces target machine code (which the locator glues into the RTOS) and the combined code (called map) gets copied into the target ROM

**Linking Process shown below:**




- The **native linker** creates a file on the disk drive of the host system that is read by a part of operating system called the loader whenever the user requests to run the programs.
- **The loader** finds memory into which to load the program, copies the program from the disk into the memory
- Address Resolution:

**Native Tool Chain:**



**Explanation for above native tool chain figure:**

- Above Figure shows the process of building application software with native tools. One problem in the tool chain must solve is that many microprocessor instructions contain the addresses of their operands.
- the above figure MOVE instruction in ABBOTT.C will load the value of variable idunno into register R1 must contain the address of the variable. Similarly CALL instruction must contain the address of the whosonfirst. This process of solving problem is called address resolution.
- When abbott.c file compiling,the compiler does not have any idea what the address of idunno and whosonfirst() just it compiles both separately and leave them as object files for linker.
- Now linker will decide that the address of idunno must be patched to whosonfirst() call instructoin. When linker puts the two object files together, it figures out idunno and whosonfirst() are in relation for execution and places in executable files.
- After loader copies the program into memory and exactly knows where idunno and whosonfirst() are in memory. This whole process called as **address resolution.**

**Output File Formats:**

In most embedded systems there is no loader, when the locator is done then output will be copied to target.

Therefore the locator must know where the program resides and fix up all memories.

Locators have mechanism that allows you to tell them where the program will be in the target system. Locators use any number of different output file formats.

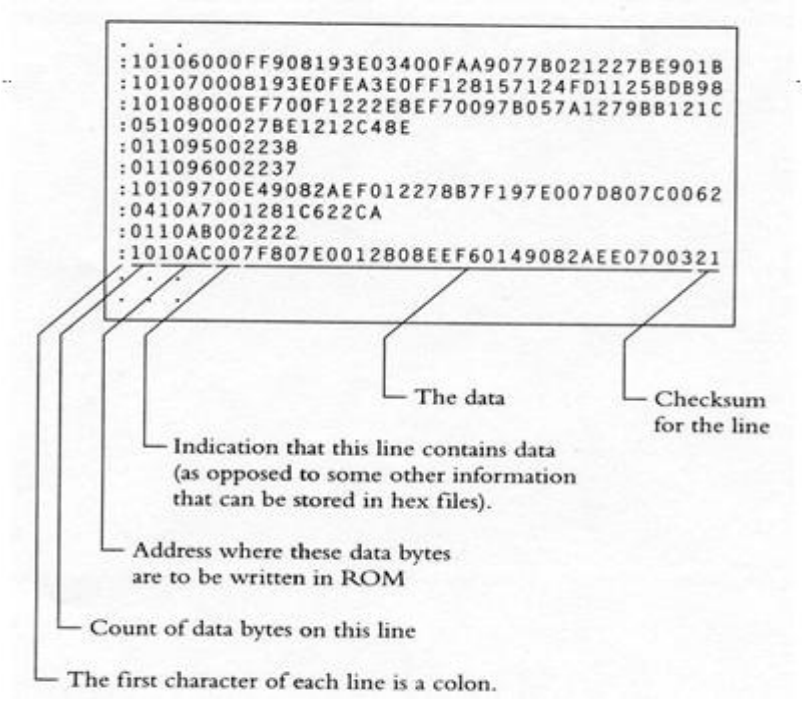
The tools you are using to load your program into target must understand whatever file format your locator produces.

**1.** intel Hex file format

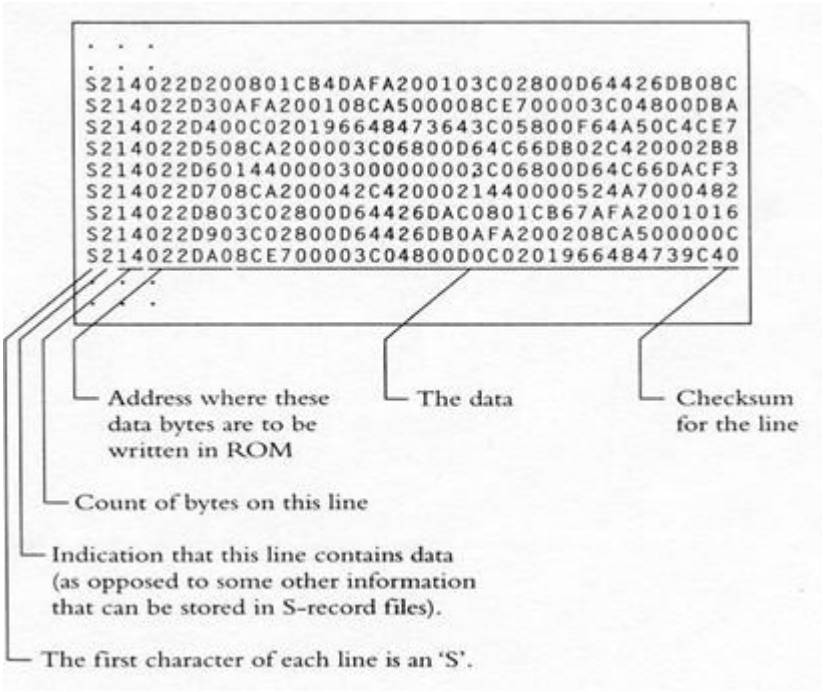
2. Motorola S-Record format

**1. Intel Hex file format:**

**below figure shows Intel Hex file format**




**2. Motorola S-Record format**



**Loading program components properly:**

Another issue that locators must resolve in the embedded environment is that some parts of the program need to end up in the ROM and some parts need to end up in RAM.

For example whosonfirst() end up in ROM and must be remembered even power is off. The variable idunno would have to be in RAM, since it data may be changed.

This issue does not arise with application programming, because the loader copies the entire program into RAM.

Most tools chains deal with this problem by **dividing the programs into segments**. Each **segment is a piece of program that the locator can place it in memory independently of other segments.**

Segments solve other problems like when processor power on, embedded system programmer must ensure where the first instruction is at particular place with the help of segments.

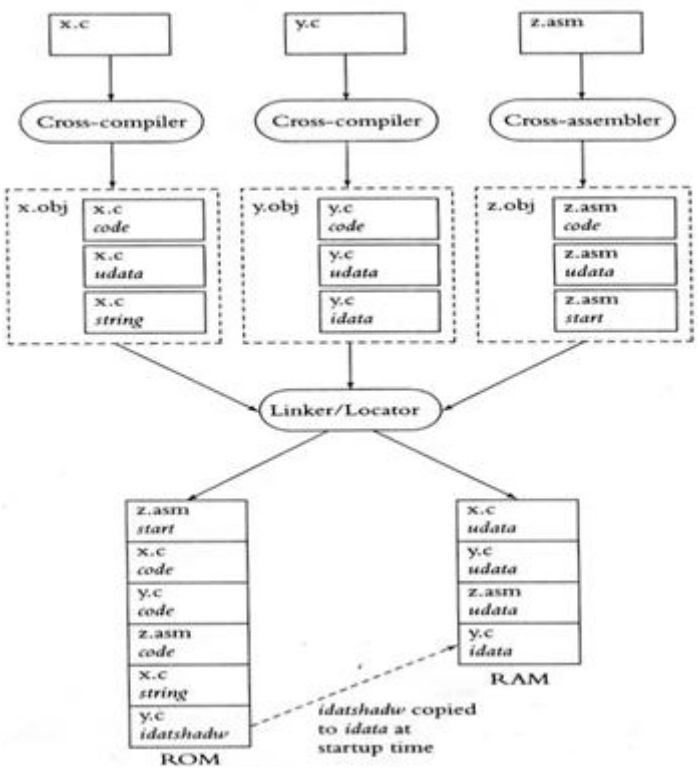



**figure: How the tool chain uses segments**

Figure shows how a tool chain might work in a system in hypothetical system that contains three modules X.c, Y.c and Z.asm.The code X.c contains some instructions, some uninitialized data and some constant strings. The Y. c contains some instructions, some uninitialized and some initialized data. The Z.asm contains some assembly language function, start up code and uninitialized code

.The cross compiler will divide X.c into 3 segments in the object file

First segment: code

Second segment: udata

Third segment: constant strings

- The cross compiler will divide Y.c into 3 segments in the object file

  First segment: code

  Second segment: udata

  Third segment: idata

- The cross compiler Z.asm divides the segments into

  First Segment: assembly language functions

  Second Segment: start up code

  Third Segment t: udata

The linker/ Locator reshuffle these segments and places Z.asm start up code at where processor begins its execution, it places code segment in ROM and data segment in RAM. Most compilers automatically divide the module into two or more segments: The instructions (code), uninitialized code, Initialized, Constant strings. Cross assemblers also allow you to specify the segment or segments into which the output from the assembler should be placed. Locator places the segments in memory. The following two lines of instructions tells one commercial locator how to build the program.

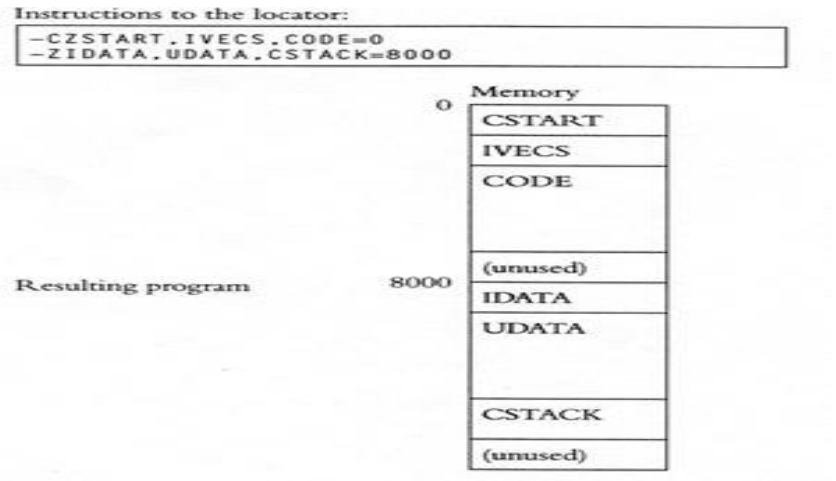



Fig 6: Locator places segments in memory

- The –Z at the beginning of each line indicates that this line is a list of segments. At the end of each line is the address where the segment should be placed.
- The locator places the segments one after other in memory, starting with the given address.
- The segments CSTART, IVECS, CODE one after other must be placed at address 0.
- The segments IDATA, UDATA AND CTACK at address at 8000.

**Some other features of locators are:**

- We can specify the address ranges of RAM and ROM, the locator will warn you if program does not fit within those functions.
- We can specify the address at which the segment is to end, then it will place the segment below that address which is useful for stack memory.
- We can assign each segment into group, and then tell the locator where the group go and deal with individual segments.

**Initialized data and constant strings:**

Let us see the following code about initialized data:

```
#define FREQ 200

Static int ifreq= FREQ;

void setfreq(int freq)

{

        int ifreq;

        ifreq = freq;

}
```

Where the variable ifreq must be stored. In the above code, in the first case ifreq the initial value must reside in the ROM (this is the only memory that stores the data while the power is off).In the second case the ifreq must be in RAM, because setfreq () changes it frequently.

The only solution to the problem is to store the variable in RAM and store the initial value in ROM and copy the initial value into the variable at startup. Loader sees that each initialized variable has the correct initial value when it loads the program. But there is no loader in embedded system, so that the application must itself arrange for initial values to be copied into variables.

The locator deals with this is to create a shadow segment in ROM that contains all of the initial values, a segment that is copied to the real initialized - data segment at start up. When an embedded system is powdered on the contents of the RAM are garbage. They only become all zeros if some start up code in the embedded system sets them as zeros.

**Locator Maps:**

- Most **locators will create an output file, called map**, that lists where the locator placed each of the segments in memory.
- A **map** consists of **address of all public functions and global variables**.
- These are useful for debugging an 'advanced' locator is capable of running a startup code in ROM, which load the embedded code from ROM into RAM to execute quickly since RAM is faster

Locator MAP IS SHOWN BELOW:

```
LINK MAP OF MODULE:  XYZ

   TYPE    BASE      LENGTH    RELOCATION   SEGMENT NAME

-----------------------------------------------------

   * * * * * * *  X D A T A   M E M O R Y  * * * * * * *

            0000H     8100H                  *** GAP ***
   XDATA    8100H     0001H     UNIT         ?XD?PROGFLSH
   XDATA    8101H     000CH     UNIT         ?XD?VPROG?PROGFLSH
   XDATA    810DH     0006H     UNIT         ?XD?CHKSM?PROGFLSH
   XDATA    8113H     0080H     UNIT         ?C_LIB_XDATA
   XDATA    8193H     0002H     UNIT         ?XD?MAIN?PAD
   XDATA    8195H     0002H     UNIT         ?XD?RXCALLBACK?PAD
    :
    :

   * * * * * * *  C O D E   M E M O R Y  * * * * * * *

            0000H     0017H                  *** GAP ***
   CODE     0080H     000FH     UNIT         PROGFLSTSTA
   CODE     008FH     0055H     UNIT         PROGFLSA
   CODE     00E4H     01ADH     UNIT         ?PR?VPROG?PROGFLSH
   CODE     0291H     0073H     UNIT         ?PR?SEND?PROGFLSH
   CODE     0304H     001DH     UNIT         ?PR?RX?PROGFLSH
   CODE     0321H     0072H     UNIT         ?PR?CHKSM?PROGFLSH
   CODE     0393H     007EH     INBLOCK      SCC_INIT
   CODE     0411H     082EH     UNIT         ?C_LIB_CODE
    :
    :
```

**Executing out of RAM:**

RAM is faster than ROM and other kinds of memory like flash. The fast microprocessors (RISC) execute programs rapidly if the program is in RAM than ROM. But they store the programs in ROM, copy them in RAM when system starts up.

The start-up code runs directly from ROM slowly. It copies rest of the code in RAM for fast processing. The code is compressed before storing into the ROM and start up code decompresses when it copies to RAM.

The system will do all this things by locator, locator must build program can be stored at one collection of address ROM and execute at other collection of addresses at RAM.

**Getting embedded software into the target system:**

- The locator will build a file as an image for the target software. There are few ways to getting the embedded software file into target system.
  - PROM programmers
  - ROM emulators
  - In circuit emulators
  - Flash
  - Monitors

**PROM Programmers:**

- The classic way to get the software from the locator output file into target system by creating file in ROM or PROM.
- Creating ROM is appropriate when software development has been completed, since cost to build ROMs is quite high. Putting the program into PROM requires a device called PROM programmer device.
- PROM is appropriate if software is small enough, if you plan to make changes to the software and debug. To do this, place PROM in socket on the Target than being soldered directly in the circuit (the following figure shows). When we find bug, you can remove the PROM containing the software with the bug from target and put it into the eraser (if it is an erasable PROM) or into the waste basket. Otherwise program a new PROM with software which is bug fixed and free, and put that PROM in the socket. We need small tool called chip puller (inexpensive) to remove PROM from the socket. We can insert the PROM into socket without any tool than thumb (see figure8). If PROM programmer and the locator are from different vendors, its upto us to make them compatible.

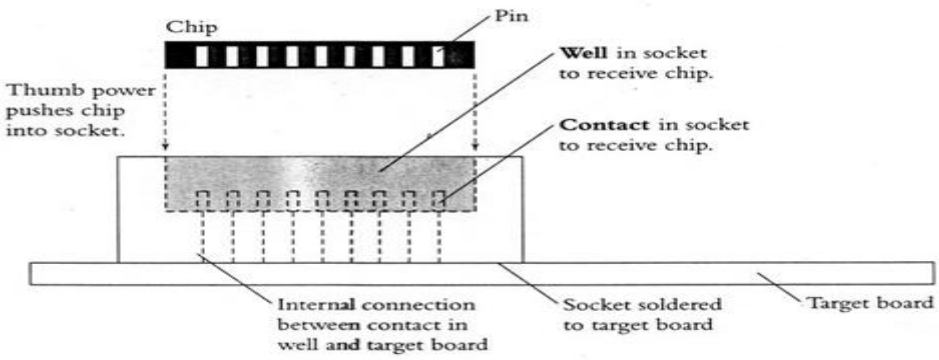



Fig : Semantic edge view of socket

**ROM Emulators:**

Other mechanism is ROM emulator which is used to get software into target. ROM emulator is a device that replaces the ROM into target system. It just looks like ROM, as shown figure9; ROM emulator consists of large box of electronics and a serial port or a network connection through which it can be connected to your host. Software running on your host can send files created by the locator to the ROM emulator. Ensure the ROM emulator understands the file format which the locator creates.

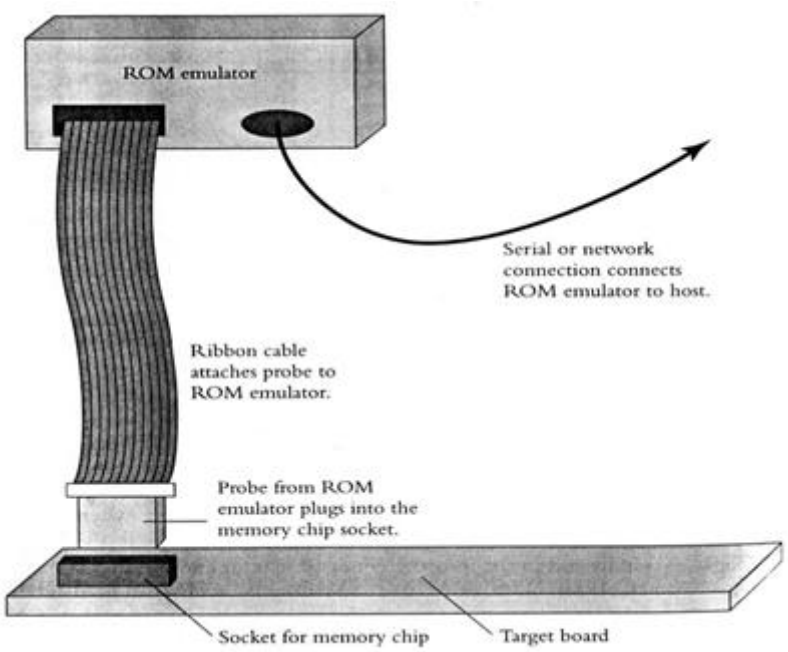



Fig: ROM emulator

**In circuit emulators:**

If we want to debug the software, then we can use overlay memory which is a common feature of in-circuit emulators. In-circuit emulator is a mechanism to get software into target for debugging purposes.

**Flash:**

If your target stores its program in flash memory, then one option you always have is to place flash memory in socket and treat it like an EPROM .However, If target has a serial port, a network connection, or some other mechanism for communicating with the outside world, link then target can communicate with outside world, flash memories open up another possibility: you can write a piece of software to receive new programs from your host across the communication link and write them into the flash memory. Although this may seem like difficult

The reasons for new programs from host:

- You can load new software into your system for debugging, without pulling chip out of socket and replacing.
- Downloading new software is fast process than taking out of socket, programming and returning into the socket.
- If customers want to load new versions of the software onto your product.

The following are some issues with this approach:

- Here microprocessor cannot fetch the instructions from flash.
- The flash programming software must copy itself into the RAM, locator has to take care all these activities how those flash memory instructions are executing.
- We must arrange a foolproof way for the system to get flash programming software into the target i.e target system must be able to download properly even if earlier download crashes in the middle.
- To modify the flash programming software, we need to do this in RAM and then copy to flash.

**Monitors:**

It is a program that resides in target ROM and knows how to load new programs onto the system. A typical monitor allows you to send the data across a serial port, stores the software in the target RAM, and then runs it. Sometimes monitors will act as locator also, offers few debugging services like setting break points, display memory and register values. You can write your own monitor program.

**DEBUGGING TECHNIQUES**

I. Testing on host machine
II. using laboratory tools
III. an example system

**Introduction:**

While developing the embedded system software, the developer will develop the code with the lots of bugs in it. The testing and quality assurance process may reduce the number of bugs by some factor. But only the way to ship the product with fewer bugs is to write software with few fewer bugs. The world extremely intolerant of buggy embedded systems. The testing and debugging will play a very important role in embedded system software development process.

**Testing on host machine :**

- **Goals of Testing process are**
  - **Find bugs early in the development process**
  - **Exercise all of the code**
  - **Develop repeatable , reusable tests**
  - **Leave an audit trail of test results**

***Find the bugs early in the development process:***

This saves time and money. Early testing gives an idea of how many bugs you have and then how much trouble you are in.

BUT: the target system is available early in the process, or the hardware may be buggy and unstable, because hardware engineers are still working on it.

***Exercise all of the code:***

Exercise all exceptional cases, even though, we hope that they will never happen, exercise them and get experience how it works.

BUT: It is impossible to exercise all the code in the target. For example, a laser printer may have code to deal with the situation that arise when the user presses the one of the buttons just as a paper jams, but in the real time to test this case. We have to make paper to jam and then press the button within a millisecond, this is not very easy to do.

***Develop reusable, repeatable tests:***

It is frustrating to see the bug once but not able to find it. To make refuse to happen again, we need to repeatable tests.

BUT: It is difficult to create repeatable tests at target environment.

Example: In bar code scanner, while scanning it will show the pervious scan results every time, the bug will be difficult to find and fix.

***Leave an "Audit trail" of test result:***

Like telegraph "seems to work" in the network environment as it what it sends and receives is not easy as knowing, but valuable of storing what it is sending and receiving.

BUT: It is difficult to keep track of what results we got always, because embedded systems do not have a disk drive.

Conclusion: Don't test on the target, because it is difficult to achieve the goals by testing software on target system. The alternative is to test your code on the host system.

**Basic Technique to Test:**

The following figure shows the basic method for testing the embedded software on the development host. The left hand side of the figure shows the target system and the right hand side shows how the test will be conducted on the host. The hardware independent code on the two sides of the figure is compiled from the same source.

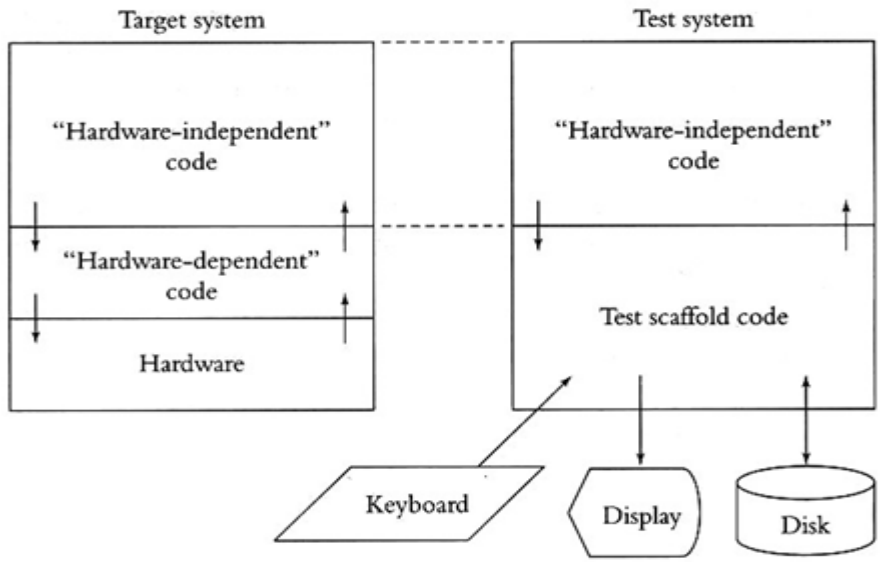



figure: Test System

The hardware and hardware dependent code has been replaced with test scaffold software on the right side. The scaffold software provides the same entry points as does the hardware dependent code on the target system, and it calls the same functions in the hardware independent code. The scaffold software takes its instructions from the keyboard or from a file; it produces output onto the display or into the log file.

**Conclusion:** Using this technique you can design clean interface between hardware independent software and rest of the code.

**Calling Interrupt Routines by scaffold code:**

Based on the occurrence of interrupts tasks will be executed. Therefore, to make the system do anything in the test environment, the test scaffold must execute the interrupt routines. Interrupts have two parts one which deals with hardware (by hardware dependent interrupt calls) and other deals rest of the system (hardware independent interrupt calls).

**Calling the timer interrupt routine:**

One interrupt routine your test scaffold should call is the timer interrupt routine. In most embedded systems initiated the passage of time and timer interrupt at least for some of the activity. You could have the passage of time in your host system call the timer interrupt routine automatically. So time goes by your test system without the test scaffold software participation. It causes your test scaffold to lose control of the timer interrupt routine. So your test scaffold must call Timer interrupt routine directly.

**Script files and Output files**:

A test scaffold that calls the various interrupt routines in a certain sequence and with certain data. A test scaffold that reads a script from the keyboard or from a file and then makes calls as directed by the script. Script file may not be a project, but must be simple one.

Example: script file to test the bar code scanner

```
#frame arrives
# Dst          Src    Ctrl
mr/56 ab
#Backoff timeout expires
Kt0
#timeout expires again
Kt0
#sometime pass
Kn2
Kn2
#Another beacon frame arrives
```

Each command in this script file causes the test scaffold to call one of the interrupts in the hardware independent part.

In response to the kt0 command the test scaffold calls one of the timer interrupt routines. In response to the command kn followed by number, the test scaffold calls a different timer interrupt routine the

indicated number of times. In response to the command mr causes the test scaffold to write the data into memory.

**Features of script files:**

- The commands are simple two or three letter commands and we could write the parser more quickly.
- Comments are allowed, comments script file indicate what is being tested, indicate what results you expect, and gives version control information etc.
- Data can be entered in ASCII or in Hexadecimal.

**Most advanced Techniques:**

These are few additional techniques for testing on the host. **It is useful to have the test scaffold software do something automatically**. For example, when the hardware independent code for the underground tank monitoring system sends a line of data to the printer, the test scaffold software must capture the line, and it must call the printer interrupt routine to tell the hardware independent code that the printer is ready for the next line.

**There may be a need that test scaffold a switch control** because there may be button interrupt routine, so that the test scaffold must be able to delay printer interrupt routine. **There may be low, medium, high priority hardware independent requests, then scaffold switches as they appear**. Some Numerical examples of test scaffold software: In Cordless bar code scanner, when H/W independent code sends a frame the scaffold S/W calls the interrupt routine to indicate that the frame has been sent. When H/W independent code sets the timer, then test scaffold code call the timer interrupt after some period. The scaffold software acts as communication medium, which contains multiple instances of H/W independent code with respect to multiple systems in the project.

**Bar code scanner Example:**

Here the scaffold software generate an interrupts when ever frame send and receive. Bar code Scanner A send data frame, captures by test scaffold and calls frame sent interrupt. The test scaffold software calls receive frame interrupt when it receives frame. When any one of the H/W independent code calls the function to control radio, the scaffold knows which instances have turned their radios, and at what frequencies.

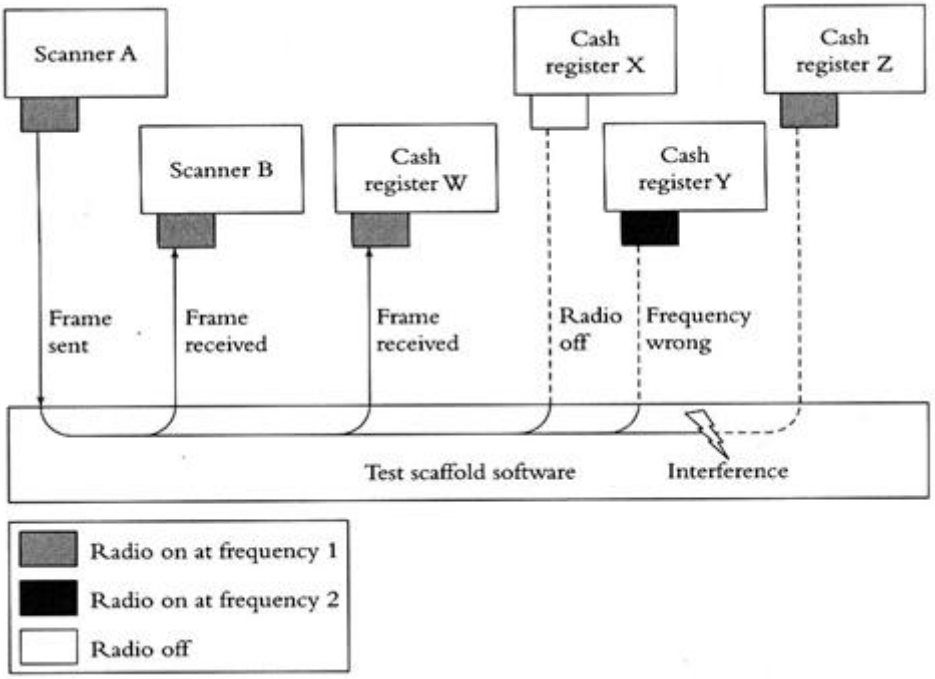



Fig2: Test scaffold for the bar- code scanner software

Targets that have their radios turned off and tuned to different frequencies do not receive the frame.

The scaffold simulates the interference that prevents one or more stations from receiving the data.

In this way the scaffold tests various pieces of software communication properly with each other or not.(see the above figure).

**OBJECTIONS, LIMITATIONS AND SHORT COMINGS:**

**Engineers raise many objections to testing embedded system code on their host system, Because many embedded systems are hardware dependent**. Most of the code which is tested at host side is hardware dependent code.

**To test at host side embedded systems interacts only with the microprocessor, has no direct contact with the hardware**. As an example the Telegraph software huge percentage of software is hardware independent i.e. this can be tested on the host with an appropriate scaffold. There are few objections to scaffold: Building a scaffold is more trouble, making compatible to RTOS is other tedious job.

**Using laboratory Tools:**

- Volt meters and Ohm Meters
- Oscilloscopes
- Logic Analyzers
- Logic Analyzers in Timing mode
- Logic Analyzers in State Mode
- In-circuit Emulators
- Getting " Visibility" into the Hardware
- Software only Monitors
- Other Monitors

**Volt meters:**

Volt meter is for measuring the voltage difference between two points. The common use of voltmeter is to determine whether or not chip in the circuit have power. A system can suffer power failure for any number of reasons- broken leads, incorrect wiring, etc. the usual way to use a volt meter It is used to turn on the power, put one of the meter probes on a pin that should be attached to the VCC and the other pin that should be attached to ground. If volt meter does not indicate the correct voltage then we have hardware problem to fix.

**Ohm Meters:**

Ohm meter is used for measuring the resistance between two points, the most common use of Ohm meter is to check whether the two things are connected or not. If one of the address signals from microprocessors is not connected to the RAM, turn the circuit off, and then put the two probes on the two points to be tested, if ohm meter reads out 0 ohms, it means that there is no resistance between two probes and that the two points on the circuit are therefore connected. The product commonly known as Multimeter functions as both volt and Ohm meters.

**Oscilloscopes:**

It is a device that graphs voltage versus time, time and voltage are graphed horizontal and vertical axis respectively. It is analog device which signals exact voltage but not low or high.

*Features of Oscilloscope:*

- You can monitor one or two signals simultaneously.
- You can adjust time and voltage scales fairly wide range.
- You can adjust the vertical level on the oscilloscope screen corresponds to ground. With the use of trigger, oscilloscope starts graphing. For example we can tell the oscilloscope to start graphing when signal reaches 4.25 volts and is rising.

Oscilloscopes extremely useful for Hardware engineers, but software engineers use them for the following purposes:

1. Oscilloscope used as volt meter, if the voltage on a signal never changes, it will display horizontal line whose location on the screen tells the voltage of the signal.
2. If the line on the Oscilloscope display is flat, then no clocking signal is in Microprocessor and it is not executing any instructions.
3. Use Oscilloscope to see as if the signal is changing as expected.
4. We can observe a digital signal which transition from VCC to ground and vice versa shows there is hardware bug.

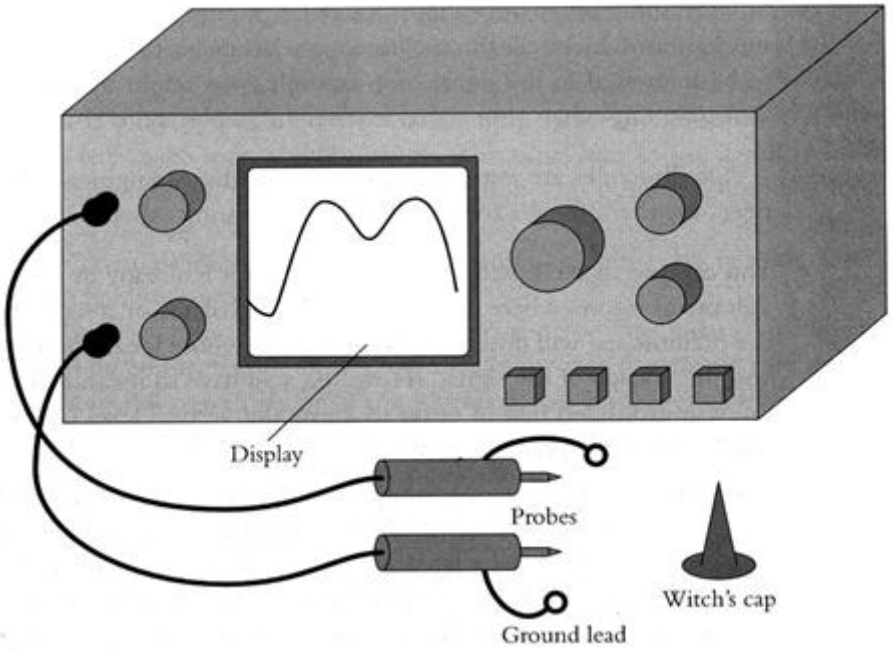
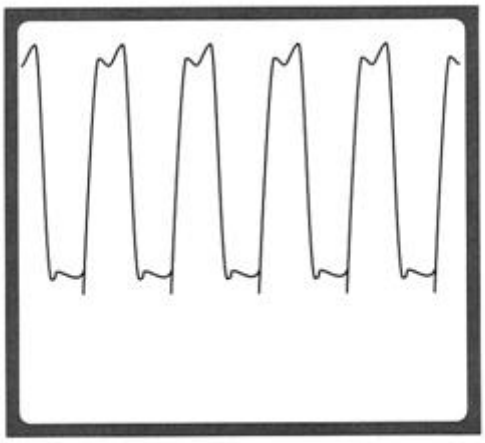
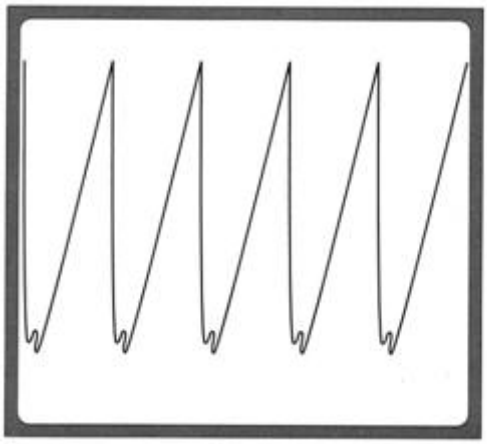



Fig3: Typical Oscilloscope

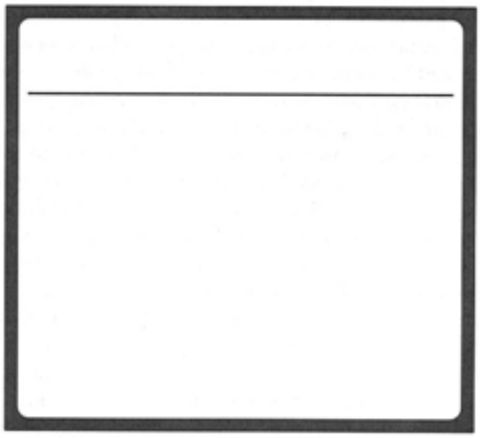
Figure3 is a sketch of a typical oscilloscope, consists of probes used to connect the oscilloscope to the circuit. The probes usually have sharp metal ends holds against the signal on the circuit. Witch's caps fit over the metal points and contain little clip that hold the probe in the circuit. Each probe has ground lead a short wire that extends from the head of the probe, it can easily attach to the circuit. It is having numerous adjustment knobs and buttons allow you to control. Some may have on screen menus and set of function buttons along the side of the screen.



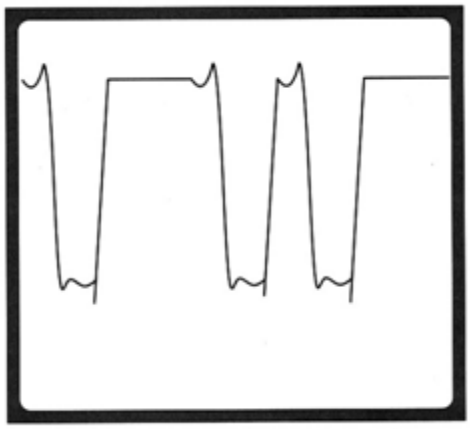
4(a): A Reasonable clock signal



4(b): A Questionable clock signal



4 (c): A dead clock signal



4(d): A ROM chip selection signal

Figure4 (a) to 4(d) shows some typical oscilloscope displays.fig (a) shows a microprocessor input clock signal. Fig (b) shows a questionable clock signal, it differs from 4(a) in that it does not go from lo to high cleanly and stay high for a period of time. Instead it draft from low to high .fig(c) shows a clock circuit that is not working at all.fig(d) shows chip enable signal.

**Logic Analyzers:**

This tool is similar to oscilloscope, which captures signals and graphs them on its screen. But it differs with oscilloscope in several fundamental ways

- A logic analyzer track many signals simultaneously.
- The logic analyzer only knows 2 voltages, VCC and Ground. If the voltage is in between VCC and ground, then logical analyzer will report it as VCC or Ground but not like exact voltage.
- All logic analyzers are storage devices. They capture signals first and display them later.
- Logic analyzers have much more complex triggering techniques than oscilloscopes.
- Logical analyzers will operate in state mode as well as timing mode.

**Logical analyzers in Timing Mode:**

Some situations where logical analyzers are working in Timing mode

- If certain events ever occur.
- Example: In bar code scanner software ever turns the radio on, we can attach logic analyzer to the signals that controls the power to the radio.
- We can measure how long it takes for software to respond.
- We can see software puts out appropriate signal patterns to control the hardware. The underground tank monitoring system to find out how long it will takes the software to turn off the bell when you push a button shown in fig5.

Example: After finishing the data transmitting, we can attach the logical analyzer to RTS and its signal to find out if software lowers RTS at right time or early or late. We can also attach the logical analyzer, to ENABLE/ CLK and DATA signals to EEPROM to find if it works correctly or not.(see fig6).

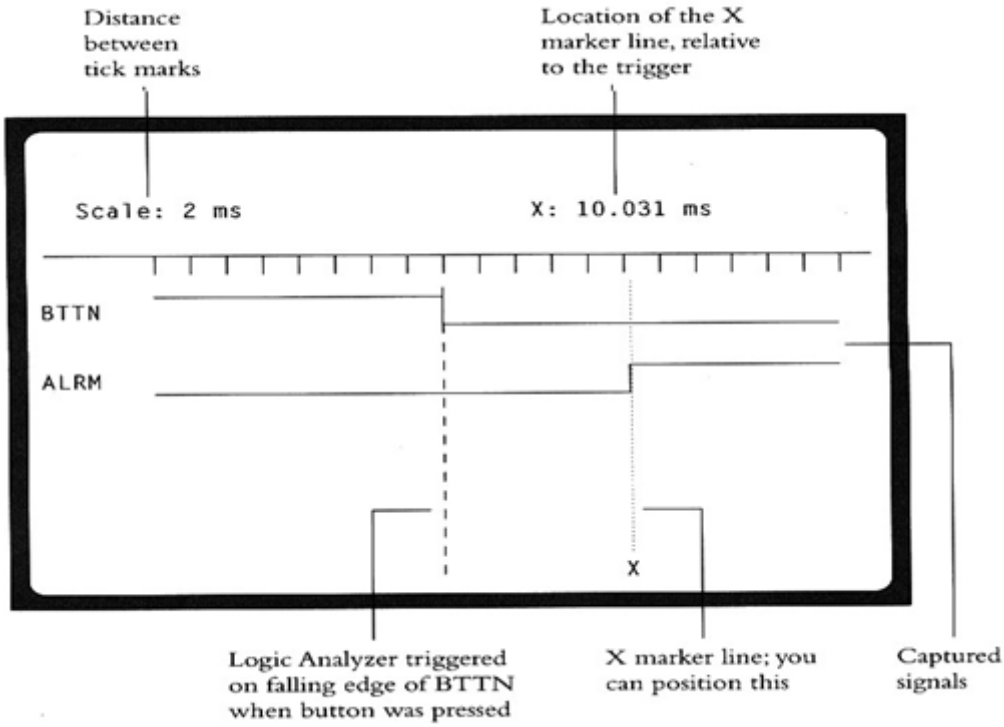



Fig5 : Logic analyzer timing display: Button and Alarm signal

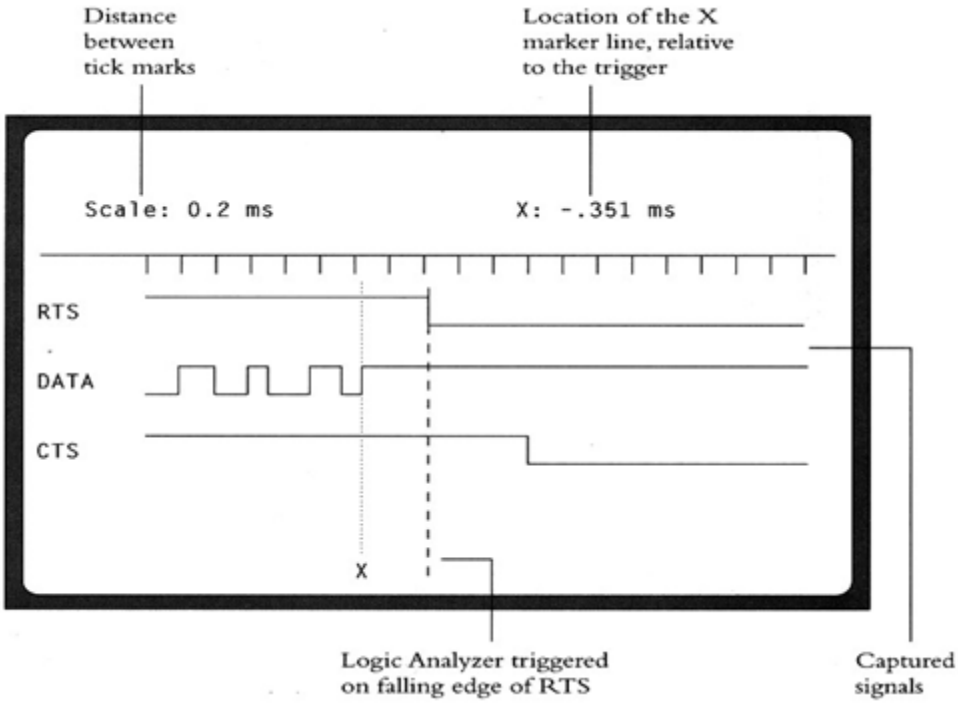



Fig6 : Logic Analyzer timing Display: Data and RTS signal

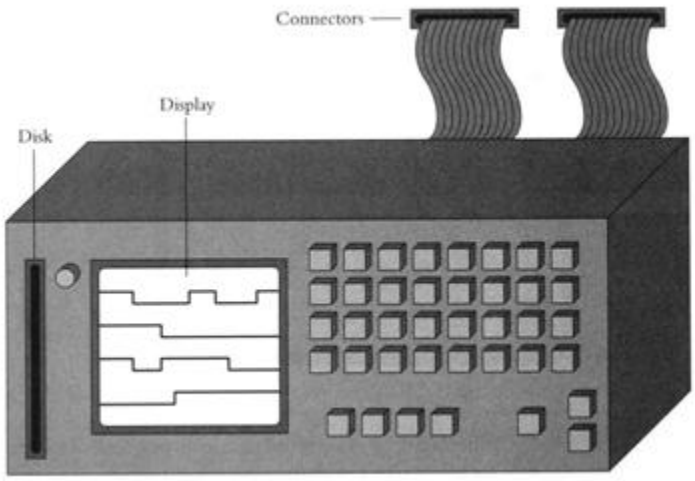



Fig7 : Logic analyzer

Figure7 shows a typical logic analyzer. They have display screens similar to those of oscilloscopes. Most logic analyzers present menus on the screen and give you a keyboard to enter choices, some may have mouse as well as network connections to control from work stations. Logical analyzers include hard disks and diskettes. It can be attached to many signals through ribbons. Since logic analyzer can attach to many signals simultaneously, one or more ribbon cables typically attach to the analyzer.

**Logical Analyzer in State Mode:**

In the timing mode, logical analyzer is self clocked. That is, it captures data without reference to any events on the circuit. In state mode, they capture data when some particular event occur, called a clock occurs in the system. In this mode the logical analyzer see what instructions the microprocessor fetched and what data it read from and write to its memory and I/O devices. To see what instructions the microprocessor fetched, you connect logical analyzer probes to address and data signals of the system and RE signal on the ROM. Whenever RE signal raise then logical analyzer capture the address and data signals. The captured data is called as trace. The data is valid when RE signal raise. State mode analyzers present a text display as state of signals in row as shown in the below figure.

| Count | Address | Data | Action | Time |
|---|---|---|---|---|
| 0001 | 13578 | 3145 | READ | 369 ns |
| 0002 | 1357A | 2241 | READ | 7.44 ns |
| 0003 | 1357C | 1199 | WRITE | 1.02 ns |
| 0004 | 1357E | 21BC | READ | 1.38 ns |
| 0005 | 02EEA | A1E3 | READ | 1.78 ns |
| 0006 | 02EEC | 1143 | READ | 2.01 ns |
| 0007 | 02EEE | BE45 | READ | 2.41 ns |
| 0008 | 02EF0 | B1B1 | READ | 2.73 ns |
| 0009 | 02EF2 | 587E | READ | 3.04 ns |
| 0010 | 02EF4 | 0032 | READ | 3.44 ns |
| 0011 | 02EF6 | 2EEE | READ | 4.01 ns |
| 0012 | 02EEE | BE45 | READ | 4.41 ns |
| 0013 | 02EF0 | B1B1 | READ | 4.73 ns |
| 0014 | 02EF2 | 587E | READ | 5.04 ns |
| 0015 | 02EF4 | 0032 | READ | 5.44 ns |
| 0016 | 02EF8 | 143A | READ | 6.04 ns |
| 0017 | 02EFA | 31BB | READ | 6.38 ns |

Fig8 : Typical logic analyzer state mode display

The logical analyzer in state mode extremely useful for the software engineer,

1. Trigger the logical analyzer, if processor never fetch if there is no memory.
2. Trigger the logical analyzer, if processor writes an invalid value to a particular address in RAM.
3. Trigger the logical analyzer, if processor fetches the first instruction of ISR and executed.
4. If we have bug that rarely happens, leave processor and analyzer running overnight and check results in the morning.
5. There is filter to limit what is captured.

**Logical analyzers have short comings:**

Even though analyzers tell what processor did, we cannot stop, break the processor, even if it did wrong. By the analyzer the processors registers are invisible only we know the contents of memory in which the processors can read or write. If program crashes, we cannot examine anything in the system. We cannot find if the processor executes out of cache. Even if the program crashes, still emulator let make us see the contents of memory and registers. Most emulators capture the trace like analyzers in the state mode. Many emulators have a feature called overlay memory, one or more blocks of memory inside the emulator, emulated microprocessor can use instead of target machine.

**In circuit emulators:**

In-circuit emulators also called as emulator or ICE replaces the processor in target system.

Ice appears as processor and connects all the signals and drives. It can perform debugging, set break points after break point is hit we can examine the contents of memory, registers, see the source code, resume the execution. Emulators are extremely useful, it is having the power of debugging, acts as logical analyzer. Advantages of logical analyzers over emulators:

- Logical analyzers will have better trace filters, more sophisticated triggering mechanisms.
- Logic analyzers will also run in timing mode.
- Logic analyzers will work with any microprocessor.
- With the logic analyzers you can hook up as many as or few connections as you like. With the emulator you must connect all of the signal.
- Emulators are more invasive than logic analyzers.

**Software only Monitors:**

One widely available debugging tool often called as Monitor .monitors allow you to run software on the actual target, giving the debugging interface to that of In circuit emulator.

Monitors typically work as follows:

- One part of the monitor is a small program resides in ROM on the target, this knows how to receive software on serial port, across network, copy into the RAM and run on it. Other names for monitor are target agent, monitor, debugging kernel and so on.
- Another part the monitor run on host side, communicates with debugging kernel, provides debugging interface through serial port communication network.
- You write your modules and compile or assemble them.
- The program on the host cooperates with debugging kernel to download compiled module into the target system RAM. Instruct the monitor to set break points, run the system and so on.
- You can then instruct the monitor to set breakpoints.

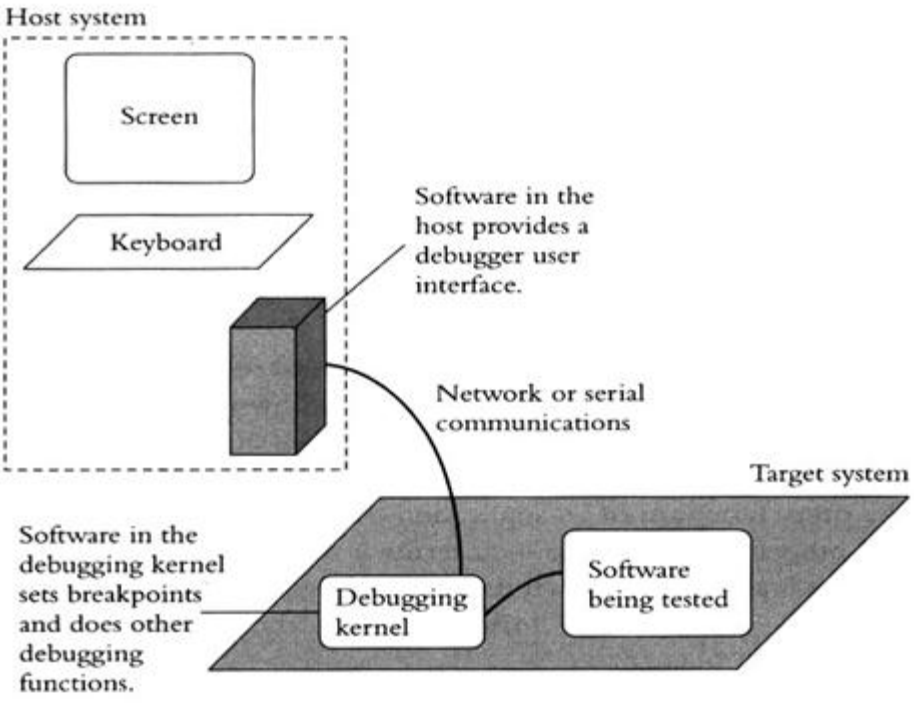



Fig 9: software only the monitor

See the above figure, Monitors are extraordinarily valuable, gives debugging interface without any modifications.

**Disadvantages of Monitors:**

- The target hardware must have communication port to communicate the debugging kernel with host program. We need to write the communication hardware driver to get the monitor working.
- At some point we have to remove the debugging kernel from your target system and try to run the software without it.
- Most of the monitors are incapable of capturing the traces like of those logic analyzers and emulators.
- Once a breakpoint is hit, stop the execution can disrupt the real time operations so badly.

**Other Monitors:**

The other two mechanisms are used to construct the monitors, but they differ with normal monitor in how they interact with the target. The first target interface is with through a ROM emulator. This will do the downing programs at target side, allows the host program to set break points and other various debugging techniques.

Advantages of JTAG:

- No need of communication port at target for debugging process.
- This mechanism is not dependent on hardware design.
- No additional software is required in ROM.