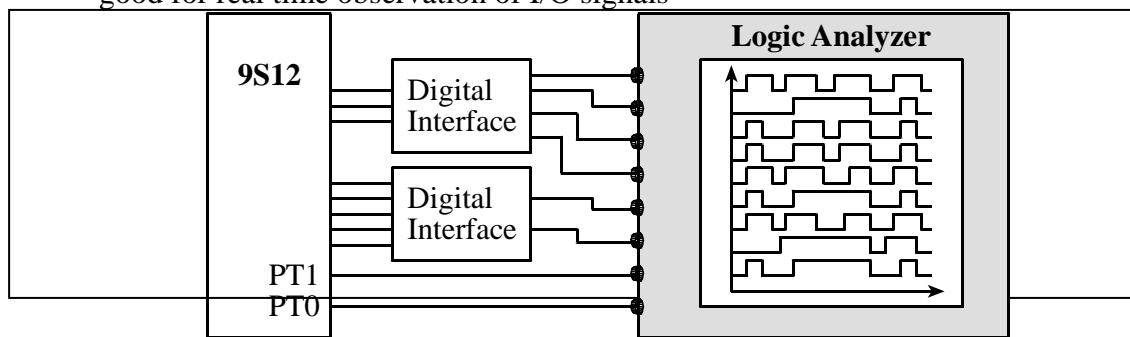


2.11.1. Debugging Tools

Software verification is a difficult but important phase.

A **logic analyzer** is a multiple channel digital storage scope

- numerous digital signals at various points in time
- attached to strategic I/O signals, real-time measurement
- attached to heart beats, profile execution
- massive amount of information
- triggering to capture data at appropriate times
- must interpret the data
- nonintrusive
- good for real time observation of I/O signals



A logic analyzer and example output.

An **emulator** is a hardware debugging tool that

- recreates the input/output signals of the processor chip
- remove the processor chip and
- insert the emulator cable into the chip socket
- operates at full speed
- observe and modify internal registers of the processor
- integrated with assembler/compiler
- expensive

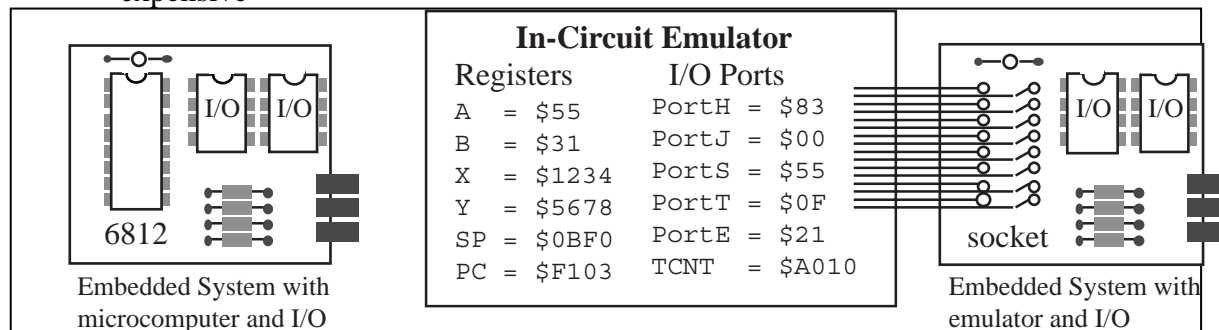


Figure 2.16. In-circuit emulator and example output.

Software-based debuggers

- breakpoint by replacing the instruction with a trap
- can not be performed when the software is in ROM

ROM-emulator.

replaces the ROM with a dual-port RAM
 while running, it fetches information from the RAM
 you can observe and modify its contents.

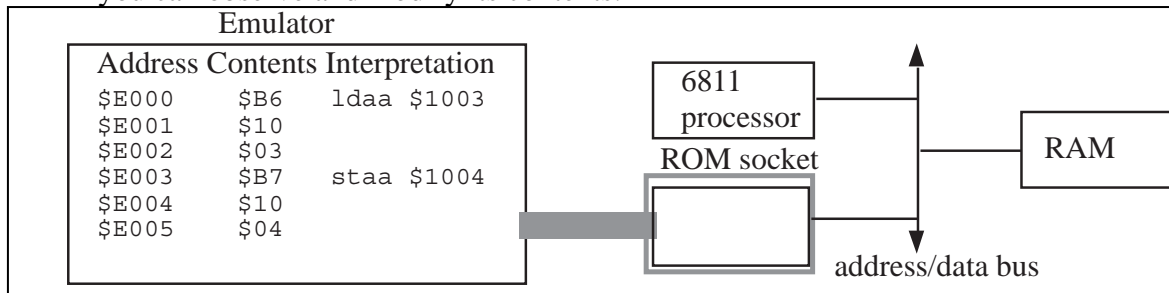
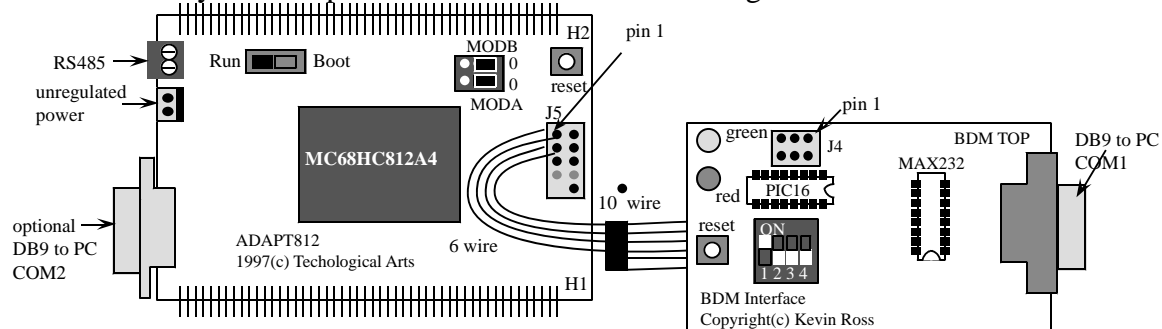


Figure 2.17. In-circuit ROM emulator and example output.

background debug module (BDM).

exists on the microcomputer chip itself
 communicates with the debugging computer
 not as flexible as an ICE,
 ability to observe software execution in real time,
 the ability to set breakpoints,
 the ability to stop the computer and
 supports hardware breakpoints.
 registers can only be observed when halted
 memory and I/O ports are accessible while running



2.11.2. Debugging Theory

“rough and ready” manual methods

desk-checking,
 dumps, and
 print statements

Debugging instrument

software code that is added for the purpose of debugging.

stabilize the system

creating a test routine that fixes (or stabilizes) all inputs.

can reproduce the exact inputs over and over again.
 modify the program,
 change in our outputs is a function of modification
 and not due to a change in the input parameters.

non-intrusive/intrusive Intrusiveness is used as a measure of the degree of perturbation caused in program performance by an instrument.

Develop your own unique debugging style.

- place all print statements in a unique column
- specific pattern in their names.
- test a run time global flag
 leaves a copy of the code in the final system
 simplifies “on-site” customer support.
- Use conditional compilation
 performance and effectiveness.

2.11.3. Functional debugging

verification of input/output parameters
 a static process where
 inputs are supplied,
 the system is run, and
 the outputs are compared against expected results.

2.11.3.1. Single Stepping or Trace

2.11.3.2. Breakpoints without filtering

2.11.3.4. Instrumentation: print statements

difficulty with print statements in embedded systems
 a standard "printer" may not be available.
 print statement itself may so slow, *intrusive*.
 print hardware used for normal operation

Appropriate debugging methods

2.11.3.5. Instrumentation: dump into array without filtering

a debugger instrument
 dumps strategic information into an array at run time.
 observe the contents of the array at a later time.
 use debugger to visualize when running.

```
short DebugList[100];
unsigned int DebugCnt=0;
void RecordIt(short data){
    if(DebugCnt==100){
        return;
    }
    DebugList[DebugCnt]=data;
```

```

    DebugCnt++;
}

```

2.11.3.6. Instrumentation: dump into array with filtering.

A filter is a software/hardware condition that must be true in order to place data into the array.

```

if(condition){
    RecordIt(MyData);
}

```

2.11.3.7. Monitor using the LED display

A monitor is an independent output process
 executes very fast, so is minimally intrusive
 small amounts of strategic information

Examples

LCD display
 LED's on individual otherwise unused output bits

2.11.4. Performance Debugging

- verification of timing behavior of our system
- a dynamic process
 system is run, and
 dynamic behavior compared to expected results

2.11.4.1. Instrumentation with independent counter

```

unsigned short Tbuf[100];
unsigned short Tcnt=0;
void RecordTime(void){
    if(Tcnt==100)
        return;
    Tbuf[Tcnt]=TCNT;    // current time
    Tcnt++;
}

```

2.11.4.2. Instrumentation Output Port.

add to ritual

```
DDRT |= 0x01; // PT0 connected to scope
```

put at beginning of function

```
PTT |= 0x01; // PT0=1
```

put at end of function

```
PTT &= ~0x01; // PT0=0
```

2.11.4.3. Measurement of Dynamic Efficiency

measure dynamic efficiency of our software.

- 1) count bus cycles using the assembly listing
too hard
- 2) uses an internal timer called TCNT.

```

unsigned short before, elapsed;
void main(void){
    ss=100;
    before=TCNT;
    tt=sqrt(ss);
    elapsed = TCNT-before-12;
}

```

Program 2.27: Empirical measurement of dynamic efficiency.

3) use an oscilloscope or a logic analyzer.

```

void main(void){
    DDRT|=0x01; // PT0 connected to scope
    ss = 100;
    while(1){
        PTT |= 0x01; // set PT0 high
        tt = sqrt(ss);
        PTT &= ~0x01; // clear PT0 low
    }
}

```

Program 2.29. Empirical measurement of dynamic efficiency.

2.11.5. Profiling

collects the time history of strategic variables

```

unsigned short time[100]; // when
unsigned short place[100]; // where
unsigned short data[100]; // what
unsigned short n;
void profile(unsigned short thePlace,
             unsigned short theData){
    if(n==100) return;
    time[n] = TCNT; // record current time
    place[n]= thePlace;
    data[n] = theData;
    n++;
}
unsigned short sqrt(unsigned short s){
    unsigned short t, oldt;
    t=0; // secant method
    profile(0,t);
    if(s>0) {
    profile(1,t);
        t=32; // initial guess 2.0
        do{
    profile(2,t);
            oldt=t; // from the last
            t=((t*t+16*s)/t)/2;}
        while(t!=oldt);}
    profile(3,t);
    return t;}

```

Program 2.30: A profile dumping into array.

2.11.5.2. Profiling using an Output Port

```

unsigned int sqrt(unsigned int s){
unsigned int t,oldt;
PTT=0;
  t=0;      // secant method
  if(s>0) {
PTT=1;
    t=32;   // initial guess 2.0
    do{
PTT=2;
      oldt=t; // from the last
      t=((t*t+16*s)/t)/2;}
      while(t!=oldt);}
PTT=3;
  return t;}

```

2.11.5.3. Thread Profile

A) One way to profile is to set bit on enter, clear bit on exit

```

void SciHandler(void){ char data;
  if(SCISR1 & RDRF){
    PTT |= 1;
    RxFifo_Put(SCIDRL); // clears RDRF
    PTT &= ~1;
  }
  if(SCISR1 & TDRE){
    PTT |= 2;
    if(TxFifo_Get(&data)){
      SCIDRL = data;} // clears TDRE
    else{
      SCICR2 = 0x2c;} // disarm TDRE
    PTT &= ~2;
  } }

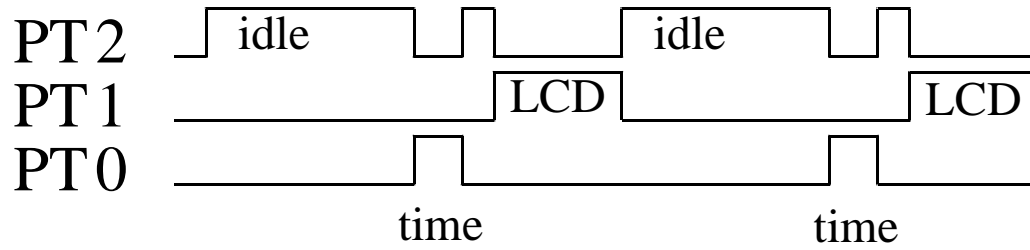
```

B) Another way to profile is to make output 1,2,4,8,...etc

```

void TimeHandler(void){ char old;
  old = PTT; PTT = 1;
  Count--; // one second yet?
  if(Count==0){
    Count = MAX;
    Sec++; NewFlag = 0;
  }
  PTT = old;
}
void main(void){
  Time_Init(RATE);
  LCD_Init();
  DDRT |= 0x07;
  while(1){
    NewFlag = 1;
    PTT = 2;
    while(NewFlag){}
    PTT = 4;
    LCD_Out(Sec);
  }
}

```



I/O bound or CPU bound??

```

unsigned short TxFifo_Size(void){
    if(TxPutPt<TxGetPt){
        return(TxPutPt+TXFIFOSIZE-TxGetPt);
    }
    else{
        return(TxPutPt-TxGetPt);
    }
}
unsigned short histogram[8];
void Collect(void){
    histogram[TxFifo_Size()]++;
}
    
```