

# Chapter 3: Hardware and Software Development Tools

## HCS12 Members

Like any other **microcontroller** family, the main differences among the HCS12 members are in their **on-chip peripheral functions**.

- **Timer** functions are very important in microcontroller applications.
  - **Input capture (IC)** function allows the microcontroller to capture the arrival time of a signal edge. **Pulse-width, period, duty cycle**, and **phase shift** can be **measured** by using this function.
  - **Output compare (OC)** function allows us to **create time delays, trigger pin actions** (toggle, pull high or low), **generate waveforms**, etc. The pulse accumulation function can be used to count the number of events occurred within a certain time interval, **measure frequency**, etc.

waveform generator
- **Pulse width modulation (PWM)** can be used to create a **digital waveform** with **duty cycles** ranges from 0 to 100 %. The variation of duty cycles can be equivalent to the change of the average magnitude of the applied voltage, such as in **DC motor speed control**.

waveform generator
- **Analog-to-Digital (A/D) Conversion** with 10-bit resolution. By using an appropriate transducer to **convert a non-electric quantity to a voltage**, data acquisition applications can be implemented with the HCS12.

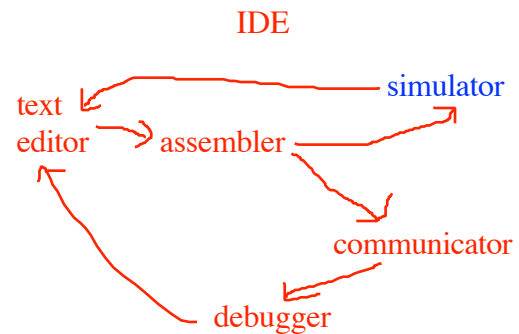
oscilloscope  
digital thermometer
- **Serial Communication Interface (SCI)** supports asynchronous serial communication. This interface is mainly used for **data communications** that utilizes the RS232 standard, for example, for communicating with a **PC** through **mouse or keyboard**.
- **Serial peripheral interface (SPI)** is a synchronous serial interface that requires a **clock signal to synchronize the data transfer** between two devices. This interface is mainly used to interface with peripheral chips, such as shift registers, seven-segment display, and LCD drivers, A/D converters, D/A converters, SRAM, and EEPROM, etc.
- **Inter-Integer Circuit (I2C)** is a serial interface standard, allowing microcontrollers and peripheral chips to exchange data.
- **Byte Data Link Communication (BDLC)** module provides access to an external serial communication multiplex bus, and operates according to the SAE J1850

protocol for low-speed ( $\leq 125\text{Kbps}$ ) data communications in automotive applications.

- The CAN 2.0 A/B protocol was primarily designed to be used as a vehicle serial data bus, meeting the specific requirements of real-time processing, reliable operation in the electromagnetic interference (EMI) environment of a vehicle, cost effectiveness, and required high bandwidth.

The MC9S12DG256 microcontroller used in this course has

- 12 Kbytes RAM
- 4Kbytes EEPROM
- 256Kbytes flash memory
- eight channels of IC and OC functions
- twenty-nine I/O ports
- SCI, SPI, five CAN, I2C, BDLC
- two 8-channel 10-bit resolution A/D converters
- 8-channel 8-bit or 4-ch 16-bit PWM port



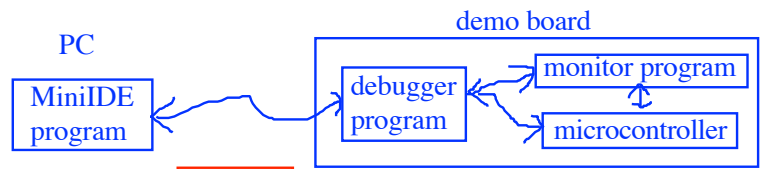
## Development Tools

Software tools include *text editors*, *communication programs*, *cross assemblers*, *cross compilers*, *simulators*, *debuggers*, and an *integrated development environment*.

(Need software to run on a PC in order to connect to a demo board with the MCU.)

- A communication program allows the user to communicate a PC with the demo board and download the program onto a demo board for execution.
- Cross assemblers and compilers translate source programs into executable machine instructions.
  - Instructions are placed in the ROM, EPROM, or the EEPROM of a MCU.
  - More than 80% of the embedded applications are written in C.
- A simulator allows the user to execute microcontroller programs in a PC without having the actual hardware.
  - It uses computer memory to represent microcontroller registers and memory locations.
  - The simulator interprets each microcontroller instruction by performing the operation required by the instruction, and then saves execution results in the computer memory.

- The source-level **debugger** is a program that allows us to **find problems in our code** at the assembly language or high-level language.
  - A debugger may have the option to **run the program on the demo board** or use a simulator.
  - A debugger may run slowly if it needs to simulate the microcontroller instructions instead of using the actual hardware to run the program.
  - A debugger can **display the contents of registers and memory** (internal and external), and program code in separate windows.
  - See the value **change of a variable** after a statement has been executed.
  - We can **set a breakpoint** at a statement.
  - A debugger needs to **communicate with the monitor program** on the demo board in order to display the contents of CPU registers and memory locations, set or delete breakpoints, trace program execution, etc.
  - Since the monitor programs on different boards may not be the same, a source-level debugger may be used only with one type of demo board.
  - The **BDM mode** of the HCS12 offers an alternative for implementing the source-level debugger.
- **Integrated development environment (IDE)** software would provide an environment that combines the above software tools so that the user could perform all development activities without exiting any one of the tools.
  - The **MiniIDE** from Mgttek combines an assembler, text editor, and a communication program.



## Demo Boards

- A **demo** (or evaluation) **board**, which **consists of microcontroller, hardware, memory, and a small monitor program**, is designed for end users to become familiar with certain microcontrollers.
- Major differences of demo boards are in the microcontroller unit and the amount of external SRAM and EPROM.
- A demo board usually contains a **small monitor program** (for **debug and monitor**):
  - Display the values of registers and memory locations.
  - Set values to registers and memory locations.
  - Set breakpoints.
  - Trace program execution.
  - Enter assembler programs directly to the demo board.
  - Disassemble the machine code on the demo board.
  - Program the on-chip EPROM, EEPROM, or flash memory.
  - Execute the program downloaded or entered to the demo board.

## Dragon12-Plus2 Trainer

- 24 MHz bus speed (generated from a 4-MHz crystal).
- D-Bug12 or serial monitor
- 16 x 2 LCD kit (4-bit interface)
- Eight LEDs
- Four seven-segment displays
- Keypad connector
- Four buttons for input
- DIP switches for input
- Buzzer for playing siren and songs (wired to the PT5 pin)
- Potentiometer for testing A/D function (wired to PAD7 pin)
- Infrared transceiver
- A small breadboard
- BDM IN and BDM OUT connectors
- Two RS232 connectors
- LTC1661 10-bit D/A converter chip with SPI interface
- 24LC16 serial EEPROM with I<sup>2</sup>C interface
- A temperature sensor

## D-Bug12 Monitor

- A monitor program facilitates the writing, evaluation, and debugging of user program.
- Provides a set of commands that allow the user to display and modify memory and register contents, download programs to the demo board for execution, set breakpoints, trace program execution, etc.

## D-Bug12 commands

- Help the program-debugging process.
- Input the commands in the **terminal window** (in MiniIDE) at the “>” prompt.

AUTO [<address>]	Enable Autostart, address is the vector
NOAUTO	Disable Autostart
BF <start address> <end address> [<data>]	Fill memory with data
BR [<address>]	Set/display user breakpoints
BR - [<address>]	Erase breakpoints
BULK	Erase entire on-chip EEPROM contents
CALL [<address>]	Call user subroutine at <address>
G [<address>]	Begin/continue execution of user code
HELP	display the MON12 command summary
LOAD [P]	Load S-records into memory, P = Paged S2
MD <start address> [<end address>]	Memory display bytes
MM <address>	Modify memory bytes (8-bit values)
MW <address>	Modify memory bytes (16-bit values)
MOVE <start address> <end address> <dest address>	Move a block of memory
RD	Display all CPU registers
RM	Modify CPU register contents
STOPAT <address>	Trace until <address>
T [<count>]	Trace <count> instructions

[ ] = optional

assembly program --> machine code  
(.asm file) (.s19 file)

### LOAD [<AddressOffset>]

- Load S-record objects into memory from an external device.
- The optional address offset is added to the load address of each S-record before its data bytes are placed in the memory.
- Providing an address allows object code or data to be loaded into memory at a location other than that for which it was assembled.
- An ASCII asterisk character is sent to the window to indicate the successful loading of each 10 S-records. Control returns to the D-Bug12 prompt.
- Load is terminated when receiving an s19 end-of-file record.
- If no s19 record is found, D-Bbug12 continues to wait for the end-of-file record unless the reset switch on the demo board is pressed.

### How to download an assembly program to a demo board?

After typing in the **LOAD** command followed by pressing the <enter> key, go back to the terminal program (**MiniIDE**) to specify the file to be downloaded by selecting the **Download File** command under the **Terminal** menu in the **MiniIDE**. After selecting the **.s19** file on the pop-up window, the file will be transferred to the demo board.

### Block Fill

**BF** <StartAddress> <EndAddress> [<Data>]

- Fill a block of memory locations with the same (optional) Data value.
- If no data field is entered, then zero will be filled in the memory locations.

> bf 1000 10FF 0

[> is the prompt on the terminal window]

set the internal memory location from \$1000 to \$10FF to zero.

**MOVE** <StartAddress> <EndAddress> <DestAddress>

- Move a block of memory from one location to another, one byte at a time.
- Memory addresses are specified in 16-bit hex values.
- The number of bytes moved is calculated by <EndAddress> - <StartAddress> + 1.
- The block of memory beginning at the destination address may overlap the memory block defined by <StartAddress> and <EndAddress>.
- One possible use is to copy a program from RAM into the on-chip.

### Memory Display

**MD** <StartAddress> [<EndAddress>]

- Display memory contents as both hexadecimal bytes and ASCII characters, 16 bytes on each line.
- If <EndAddress> is not provided, only a single line is displayed.
- The number supplied as the <StartAddress> parameter is rounded down to the next lower multiple of 16, while the number supplied as the <EndAddress> parameter is rounded up to the next higher multiple of 16 minus 1. This causes each line to display memory in the range of \$xxx0 to \$xxxF.

> md 1000

displays one line; the actual memory range displayed is \$1000 through \$100F.

> md 1005 1020

displays three lines; the actual memory range displayed is \$1000 through \$102F.

### Memory Modify

**MM** <Address> [<Data>]

- Examine and modify the contents of memory locations one byte at a time.
- If the 8-bit data parameter is present on the command line, the byte at memory location <Address> is replaced with <Data> and the command is terminated.
- If no optional data is provided, D-Bug12 will enter the interactive memory modify mode:
  - Each byte is displayed on a separate line following the address of data.
  - A single character sub-command can be entered on the same line before hitting carriage return.
  - The sub-commands have the following formats.

<enter> = press the “enter” key on the keyboard.

[<Data>] <enter>	update current location and display the next location
[<Data>] </> or <=>	update current location and redisplay the same location
[<Data>] <^> or <->	update current location and redisplay the previous location
[<Data>] < . >	update current location and exit Memory Modify Mode

Example: Note that **each line is terminated** with pressing the <enter> key on the keyboard, although it is not shown in the following sequence.

```
> mm 1000          ; display the contents of memory location $1000
1000 00           ; show 00 as the content; press <enter> to next memory location
1001 00 FF        ; show the content of $1001; change it to FF and press <enter>
1002 00 ^         ; use ^ to show (previous location) $1001 in next line
1001 FF           ; display the new content of $1001; press <enter> to next location
1002 00           ;
1003 00 55 /      ; change $1003 to 55 and redisplay the content again in next line
1003 55 .         ; exit the memory modify mode
>                ; back to the debugger
```

#### **MW <Address> [<Data>]**

- Same as the MM instruction, but for 16-bit hex data (**2 byte at a time**).
- The memory location is incremented or decremented by 2, instead of one.

Example: Note that **each line is terminated** with pressing the <enter> key in the keyboard, although it is not shown in the following sequence.

```
> mw 1100          ; display the contents of memory location $1100
1100 00F0          ; show the content; press <enter> to move to next memory location
1102 AA55 0008     ; display the content of $1102; change it to 0008; press <enter>
1104 0000 ^        ; use ^ to show $1102 in next line
1102 0008          ;
1104 0000          ;
1106 0000 .        ; exit the memory modify mode
>                ; back to the debugger
```

**Breakpoint** **BR** [<Address> <Address> ...]

- Set a **breakpoint** at a **specified address** or displays any previously set breakpoints.
- The function of a breakpoint is to halt user program execution when the program reaches the breakpoint address.
- When a breakpoint address is encountered, D-Bug12 disassembles the instruction at the breakpoint address, prints the CPU **register contents**, and **waits** for a D-Bug12 **command** to be entered by the user.
- Entering **BR** without any addresses will **display** all the currently set **breakpoints**.
- [Useful for **debugging**] When our program is not working correctly and we suspect that the instruction at certain memory location is incorrect, we can set a breakpoint at that location and check the execution result at the CPU registers and memory locations.

**Breakpoint Remove** **NOBR** [<Address> <Address> ...]

- **Removes** one or more previously entered breakpoints.
- Entering **NOBR** – (without any addresses) removes all the currently set breakpoints.

**Goto** **G** [<Address>]

- Begin **execution of user program** in **real time** from the optional address.
- If optional address is not provided, the execution begins at the address defined by the present value of PC.
- Execution of the user program continues **until** a user **breakpoint** is encountered, a CPU **exception** (= **fatal error**) occurs, the **STOP** or **RESET (SWI)** command is entered, or the demo board's **reset switch** is pressed.
- When the **program halts**, control is **returned to D-Bug12** and a message is displayed explaining the reason for user program termination. The instruction at the current **PC address** and the CPU **register contents** will be displayed.

**Trace** **T** [<count>]

- [Useful for **debugging**] **Trace** command is used to **execute one or more** user program **instructions** beginning at the **current PC location**.
- As each instruction is executed, the CPU **register contents** are displayed.
- **Branch** instructions (BCC, LBCC, BRSET, BRCLR, DBEQ/NE, IBEQ/NE, TBEQ/NE) that contain an offset that branches back to the instruction opcode do not execute.
- D-Bug12 appears to become stuck at the branch instruction and does not execute the instruction. This limitation can be overcome by using the GT command to set a temporary breakpoint at the instruction following the branch instruction.



Example:

```
> br 1010 ; set breakpoint
Breakpoints: 1010 0000 0000 0000 [this line appears in the Terminal window]
```

```
> g 1000 ; execute the program and stop at $1010
PC    SP    X    Y    D=A:B    CCR= SXHI NZVC
1010  0A00  0032  0900    00:31    1001 0000
```

```
> t ; trace 1 instruction = single step
PC    SP    X    Y    D=A:B    CCR= SXHI NZVC
1012  0A00  04D2  0900    30:39    1001 0000
```

```
> t 2 ; trace 2 instructions
PC    SP    X    Y    D=A:B    CCR= SXHI NZVC
1016  0A00  04D2  0900    30:39    1001 0000
```

```
> nobr - ; erase all breakpoints
Breakpoints: 0000 0000 0000 0000
```

### Call [<Address>]

- **Execute a subroutine** and return to the D-Bug12 monitor program when the final **RTS** of the subroutine is executed.
- All CPU **registers contain** the values at the time the subroutine is returned, with the exception of PC, which still contains the starting address of the subroutine.
- If a subroutine address is not supplied on the command line, the current value of PC is used as the starting address.

### Register Display

**RD** is used to **display** the contents of the CPU registers (PC, SP, X, Y, A, B, CCR).

### Register Modify

#### **RM**

- **Examine or modify** the contents of the CPU registers (PC, SP, X, Y, A, B, CCR) .
- When the content of each register is displayed, the debugger allows the user to enter a new hex.
- Press the <enter> key without first typing any new hex value will **display** the content of **next register**.
- Typing a **period** on a line will **exit** the interactive mode.

```
> rm ; type the command and press <enter>
PC=0000 1000 ; display PC's content; change to 1000; press <enter>
SP=0A00 ; display SP's content; press <enter>
IX=0000 0100 ;
```

```

IY=0000      ;
A=00         ;
B=00 FF      ;
CCR=90 D1    ;
PC=1000 .    ; display PC again; exit the debugger
>

```

### <RegisterName> <RegisterValue>

- **Change** the **value** of any **CPU registers** (PC, SP, X, Y, A, B, D, CCR).
- Each of the field in the CCR may be modified by using the bit names.

CCR Bit Name	Description	Legal Values
S	Stop enable	0 or 1
H	Half carry	0 or 1
N	Negative flad	0 or 1
Z	Zero flag	0 or 1
V	2's complement over flag	0 or 1
C	Carry flag	0 or 1
IM	IRQ interrupt mask	0 or 1
XM	XIRQ interrupt mask	0 or 1

Example:

```

>pc 2000
PC    SP    X    Y    D=A:B    CCR= SXHI NZVC
2000  0A00  0100  0000    00:FF    1101  0001

```

```

>x 800
PC    SP    X    Y    D=A:B    CCR= SXHI NZVC
2000  0A00  0800  0000    00:FF    1101  0001

```

```

>x 0
PC    SP    X    Y    D=A:B    CCR= SXHI NZVC
2000  0A00  0800  0000    00:FF    1101  0000

```

```

>z 1
PC    SP    X    Y    D=A:B    CCR= SXHI NZVC
2000  0A00  0800  0000    00:FF    1101  0100

```

```

>d 2010
PC    SP    X    Y    D=A:B    CCR= SXHI NZVC
2000  0A00  0800  0000    20:10    1101  0100

```

# The Memory Map

Table 3.2 D-Bug12 memory map for HCS12Dx256

Address range	Description
\$0000-\$03FF	I/O registers
\$0400-\$0FFF	on-chip EEPROM
\$1000-\$3BFF	on-chip SRAM (available to user)
\$3C00-\$3FFF	on-chip SRAM (D-Bug12)
\$4000-\$EE7F	D-Bug12 code
\$EE80-\$EEBF	User accessible function table
\$EEC0-\$EEFF	Customization data
\$EF00-\$EF8B	D-Bug12 startup code
\$EF8C-\$EFFF	Secondary reset/interrupt table
\$F000-\$FFFF	Bootloader

- The D-Bug12 monitor operates from the flash memory.
- **\$0000~\$03FF** are reserved for internal **peripheral registers**. Any access to addresses in this memory space will be directed to the peripheral registers.
- The users are restricted to use SRAM (from **\$1000 to \$3BFF**) or EEPROM (upper 3K bytes) to run **application programs**.
- \$4000-\$EE7F are reserved by the D-Bug12 monitor program.
- \$EF00~\$EF8B are reserved by the D-Bug 12 monitor for Autostart. \$0FF0-\$0FFF are also reserved.
- A good choice is to use the memory space:
  - **\$1000~\$1FFF hold the data storage and stack**. Be careful not to overlap the data with the stack.
  - **\$2000~\$3BFF are better for application program**.


## Using the Dragon12-Plus2 Trainer

1. Download MiniIDE from [www.mgtek.com/miniide](http://www.mgtek.com/miniide) if the computer does not have the software.
2. Connect the **USB cable** to the demo board.
3. **Start** the **MiniIDE** program by double-clicking its icon on the computer screen. Click on the **Terminal** menu and select **Option**. Check that the **port number** is the same as the one written on the box of the demo board.
4. Click on the **File** menu from the MiniIDE window and select **New** for **typing** in a **new program** file (e.g., in [p. 13](#)) or select **Open** for editing an existing program file. Afterwards, type or edit the object code on the editor window on the right. **Save** the file with file type of **“all”** and file name with the **“.asm”** extension
5. To **assemble** the program, click the **Build** menu and select the desired **“.asm”** file. The status window at the bottom of the MiniIDE window displays the status of the assembler. If there are errors, correct the object code in the editor window. [follow step 2 in p. 13]
6. To communicate with the demo board. If the terminal window is not on the left of the MiniIDE window, click on the **Terminal** menu and select **Show Terminal Window**. Press the **RESET** button on the demo board to initialize the terminal window on the computer. A message will be displayed on the terminal window and prompt you to **“press key to start monitor...”** Now, **press any key** on the keyboard and the monitor will be activated. The **“>”** prompt should be shown on the terminal window. [follow step 3 in p. 14]
7. Download the object code by first typing the **LOAD** command followed by **pressing** the **<enter>** key in the terminal window. Then, select the **Download File** command under the **Terminal** menu to select the downloading **“.s19”** file.
8. To execute the program by typing **G 2000** and press the **<enter>** key in the terminal window and check the results to see if the program is correct. (Note: 2000 is the start address of the execution part of your program.) [follow step 5 in p. 14]
9. If necessary, use the D-Bug12 commands to debug, set any breakpoints, etc. [follow other steps in pp. 14-16]
10. Repeat steps 6-9 every time you need to download a new or edited program.

## Case Study: Convert Hex to BCD ASCII Digits

1. Type the following sequence in the editor window of MiniIDE and save as *example.asm* in the directory of your choice (usually ENNG36 in C drive).

### initialization

a label points to address \$1000 → data	org	\$1000		
	fdb	12345	; value to be converted to ASCII code	
a label points to address \$1100 → result	org	\$1100		1 --> \$31
	rmb	5		2 --> \$32
				3 --> \$33
	org	\$2000	; starting address of the program	4 --> \$34
	ldd	data	; place the value in D	5 --> \$35
	ldy	#result		
	ldx	#10	; divide-by-10 to isolate each digit	
	idiv		; "	
	addb	#\$30	; add \$30 to B to convert to ASCII code	
	stab	4,Y	; store the ASCII code of the last digit	
	xgdx		; swap the quotient to D. (This is the first xgdx)	
	ldx	#10		
	idiv			
	addb	#\$30		
	stab	3,Y	; store the ASCII code of the 4th digit	
	xgdx			
	ldx	#10		
	idiv			
	addb	#\$30		
	stab	2,Y	; store the ASCII code of the 3rd digit	
	xgdx			
	ldx	#10		
	idiv			
	addb	#\$30		
	stab	1,Y	; store the ASCII code of the 2nd digit	
	xgdx			
	addb	#\$30		
	stab	0,Y	; store the ASCII code of the 1st digit	
	swi			
	end			

2. To compile the program, click on the **Build** menu and select *Build example.asm*. Make sure that there is **no error message** in the status window. The output of the assembly process is an S-record file, *example.s19*.

3. Press the **RESET** button on the demo board. Press <enter> on the keyboard. Type **load** and press <enter> at the ">" prompt of the terminal window. Click on the **Terminal** menu and select **Download File** to download *example.s19*.
4. Before running the program, you can verify that the program data is downloaded correctly into the memory.
  - Type **md 1000** in the terminal window. The contents at \$1000 and \$1001 displayed in terminal window should be **\$3039**. This corresponds to the **hex** version of the decimal number 12345 stored in the label *data*.
5. Now **execute the program** without setting any breakpoints and check the results to see if it is correct. Type **g 2000** in the terminal window.
  - The **termination of the programs** is caused by the **software interrupt (swi)** instruction, which is often used as the last instruction of a program to **return the control to the D-Bug12 monitor**.
  - The following registers' contents should be found in the terminal window.

>g 2000

PC-2036    Y-1100    X-0032    A-00    B-31    C-D0    S-3E5E    PG-00  
>

6. Type **md 1100** to **see the contents** of the memory locations at \$1100 to \$1104, representing the ASCII codes of 1, 2, 3, 4, and 5, as in the following. If yes, the program has been executed correctly.

>md 1100

1100	31	32	33	34	35	00	00	...	00	12345
1110	00	00	00	00	00	00	00	...	00	
1120	...									
...										
1180	00	00	00	00	00	00	00	...	00	

>

7. To show how to **correct a program mistake**, let **comment out the first XGDx** instruction in the program by inserting a ";" in front of XGDx.
  - Repeat steps 1-3 to save and recompile the .ASM file, and download the .S19 file.
  - Type **g 2000** and press <enter>. Then, type **md 1100** and press <enter>. Now, the **first four digits** (i.e., 0053) are **incorrect**.

>md 1100

1100	30	30	35	33	35	00	00	...	00	00535
1110	...									

8. We can **trace the execution of instructions** in the program by **setting breakpoints** at locations that we have suspicious about.

- Step 1: Set a breakpoint **at \$200F** so that we can trace through **first six instructions** and check to see if the quotient (stored in X) and the memory contents at \$1104 are correct.
- Type **br** and press <enter> to see up to four breakpoints. Type **br 200F** and press <enter> to set the first breakpoint to location \$200F.
  - Type **g 2000** and press <enter> to run the program again. When the program stops, type **md 1100** and press <enter> to inspect.

>g 2000

PC-200F    Y-1100    X-04D2    A-00    B-35    C-D0    S-3E5C    PG-00

>md 1100

1100   30     30     35     33     35     00     00     ...     00     00535  
1110   ...

- X contains hex value \$04D2 (= decimal 1234) and is the correct quotient.
- B and \$1104 contain hex value \$35 and is the ASCII code of 5.
- In the next division, the number to be divided by 10 will be 1234.
- The next instruction is LDX #10

Step 2: Trace the instructions by typing **T** and press <enter>.

> T

PC-2012    Y-1100    X-000A    A-00    B-35    C-D0    S-3E5B    PG-00

> T

PC-2014    Y-1100    X-0005    A-00    B-03    C-D0    S-3E5A    PG-00

>

- After the first **T**, D=A:B does **not contain the value of 1234**, rather it contains \$0035 before the IDIV instruction is executed. It is because we **forgot to swap the value in X with D** (by commenting out the first XGDX) before performing the second division!

Step 3: **Fix the problem by deleting the “;”** in the front of the first XGDX instruction before the LDX #\$0A instruction. **Save, recompile, download, and re-run** the program by following steps 1-5. (In case, old breakpoints are still present, use **nobr** – to erase them.)

```

>load
*
>g 2000

PC-2036   Y-1100   X-0032   A-00   B-31   C-D0   S-3E5E   PG-00
>md 1100

1100  31    32    33    34    35    00    00    ...    00    12345
1120  00    00    00    00    00    00    00    ...    00
...
1180  00    00    00    00    00    00    00    ...    00
>

```

- Now \$1100-\$1104 contains the correct values.

End of Chapter 3