Chapter VII: Firmware

I. Introduction

The AES firmware is the set of all hex codes in the ROM chip (U3). These codes include BASIC 11, the AES MONITOR, the DEBUGGER, and the small utility programs used to control the hardware on the AES-11 board. Most of these subjects are covered in chapter V. Here we discuss the collection of utility routines which were written in assembly language and placed into ROM.

II. AES Utility Routines

AES utility routines is a set of firmware programs designed to aid you in using on board devices like the LCD etc. These routines can be called from Basic or assembly language programs. See the AES11ROM.ASM on the AES-11 disk for the utility routines source code.

Key Template:

0	1	2	3	ESC
4	5	6	7	TEST
8	9	A	B	RUN
C	D	E	F	FUNC

B. Keypad and LCD routines

1. **Function: LCDCLS**
   DEBUGGER: B 9000
   BASIC11: x=call($9000)
   Assembly: JSR $9000

   Clears the LCD screen and sets the LCD cursor to location 0

2. **Function: LCDCHAR**
   DEBUGGER: B 9003
   BASIC11: x=call($9003)
   Assembly: JSR $9003

   Sends ASCII code in memory 6040H to the LCD.

3. **Function: KEYCHAR**
   DEBUGGER: B 9006
   BASIC11: x=call($9006)

Assembly: JSR $9006

Gets **ASCII** input equivalent to the (0-F) key on the keypad and stores in memory 6040H. Program will not exit until a key (0-F) has been pressed. Displayable key codes will be send to LCD.

4. **Function: KEYHEX**
DEBUGGER: B 9009
BASIC11: x=call($9009)
Assembly: JSR $9009

Gets **hexadecimal** input (0-Fh) equivalent to the (0-F) key on the keypad and stores in memory 6040H. Program will not exit until a key (0-F) has been pressed. Displayable key codes will be send to LCD.

5. **Function: KEYNUM**
DEBUGGER: B 900C
BASIC11: x=call($900C)
Assembly: JSR $900C

Gets **numeric** input (0-9) equivalent to the (0-9) key on the keypad and stores in memory 6040H. Program will not exit until a key (0-9) has been pressed. Displayable key codes will be send to LCD.

6. **Function: KEYCODE**
DEBUGGER: B 900F
BASIC11: x=call($900F)
Assembly: JSR $900F

Gets the hardware key code (0-19) and stores it in memory 6040H.

0	4	8	12	16
1	5	9	13	17
2	6	10	14	18
3	7	11	15	19

7. **Function: SETCUR**
DEBUGGER: B 902D
BASIC11: x=call($902D)
Assembly: JSR $902D

Sets LCD cursor location to be same as the number in memory **6041H**. Cursor location must be from 0 to 31.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

B. Serial I/O

1. **Function: GETCHRE**
   DEBUGGER: B 9012
   BASIC11: x=call($9012)
   Assembly: JSR $9012

   Receives byte from serial port at existing setup and puts it in memory 6040H. Character will be sent to the CRT display.

2. **Function: GETCHR**
   DEBUGGER: B 9015
   BASIC11: x=call($9015)
   Assembly: JSR $9015

   Receive byte from serial port at existing setup and puts it in memory 6040H. Character will be sent to the CRT display.

3. **Function: PUTCHR**
   DEBUGGER: B $9018
   BASIC11: x=call($9018)
   Assembly: JSR $9018

   Sends **ASCII** code in 6040H to the CRT display through the 1st serial port

4. **Function: ENABLES2**
   BASIC11: x=call($903C)
   Assembly: JSR $903C

   Enable the 2nd serial port. I/O input will be discontinued from the 1st serial port.

5. **Function: ENABLES1**
   Assembly: JSR $903F
   BASIC11: x=call($903F)
   Enable the 1st serial port.

C. A/D and D/A conversion

1. **Function: ADPE0**
   DEBUGGER: B $9021
   BASIC11: x=call($9021)
   Assembly: JSR $9021

Starts A/D (analog to digital) conversion at PORT E pin 0 and stores result in memory 6040H

2. **Function: DAC**
DEBUGGER: B 9020
BASIC11: x=call($9020)
Assembly: JSR $9020

Reads value D (digital) in memory 6040H and makes D/A (digital to analog) conversion to the D/A output.

D. Other special functions

1. **Function: CRTTEXT**
Assembly: JSR $9027

Sends text string ending with null character to the CRT display indexed by IX register.

```
ldx   #mesg
jsr   $9027
```

2. **Function: LCDTEXT**
Assembly: JSR $902A

Sends text string ending with null character to the LCD indexed by IX register.

```
ldx   #mesg
jsr   $902A
```

3. **Function: LEDSON**
DEBUGGER: B 901B
BASIC11: x=call($901B)
Assembly: JSR $901B

Toggle six leds on the AES-11 board on.

4. **Function: LEDSOFF**
DEBUGGER: B 901E
BASIC11: x=call($901E)
Assembly: JSR LEDSOFF or JSR $901E

Toggle six leds on the AES-11 board off.

5. **Function: SQWAVE**
DEBUGGER: B $9030
BASIC11: x=call($9030)
Assembly: JSR $9030

Generate a 50Hz (default) square wave at PORT A pin 6 running continuously.

6. **Function SQWAVFF**
DEBUGGER: B $9033
BASIC11: x=call($9033)
Assembly: JSR $9033

Stop generating a square wave at PORT A pin 6

7. **Function: S19UPLOAD**
DEBUGGER: U
BASIC11: x=call($9039)
Assembly: JSR $9039

Allows the user to download a Motorola hex file onto the AES-11 board and load the program into AES-11 RAM memory starting address specifies in the hex file.

8. **Function: AUTORUN**
DEBUGGER: B 9042
Assembly: JSR $9042

This function will place a 1 in memory 6052Hex to indicate a self-start program. Upon power up, the DEBUGGER will execute the program stored in RAM starting at 100Hex.

III MC68HC11 Floating-Point Package

Suppose you have the result of a 12 bit A/D conversion, say 9B3Hex, stored in two bytes of RAM (only 3 nibbles are required). Further suppose it is required to multiply this result by 3.8884decimal to get the needed result, which may be pressure or temperature etc. You will have to write assembly language subroutines to remove the decimal, convert all numbers to hex and do the multiplication. And then convert the result to decimal (it is 96548972decimal) and put the decimal point back in to get a final result (9654.8972decimal) so that it can be printed out or stored in a disk file in an understandable format. This is hexadecimal integer (fixed-point) math.

We show next how to use the floating-point routines in the AES-11 ROM. It requires a little work, but it is still preferable to the above method in many cases.

The assembly source code for the floating-point routines is in the 8-Bit MCU Applications manual referenced in Appendix A1, and it is also on the Motorola WEB site. The hex code in ROM required for the floating-point routines takes up about 2500 bytes of memory. If you needed only one routine, say divide, then far less memory would be required.

Some math operations, like square root, take one variable and some, like multiplication, take two. Two 5-byte memory locations ($6100 through

$6109 in RAM) are set aside for the variables to be operated on by the math routines. When we perform a multiplication two variables are taken from the two memory locations, multiplied, an then the result is put into one of the same two locations. The numbers in these two 5-byte locations are in a compacted form that is written and read only by the floating-point routines.

Your numbers can be input in one of two forms. First is a hexadecimal integer, like 55AF3Hex, and the second is a ascii string, like -3.9945E+12. In the case of ascii strings, you can use any reasonable way or writing them. A few examples: 21.5567, 0.0033445, -222.333, or 345.9993E-7. The only restriction is the ascii string must fit into 14 or fewer bytes. Each number, decimal point, sign, and E will take one byte. A 14 byte ascii number is +23.997345E+13 . The maximum size number is 8 digits, like 3.1415927 or like the 14 byte example. A floating-point routine will compact this number into a 5 byte floating-point format.

An easy example of how a hex integer ends up in your control system is the A/D converter. The converters on the 68HC11A0 measure a voltage and place a hex value between 00H and FFH in one of the A/D Result Registers, ADR1 through ADR4. ASCII requires more explanation. The ascii number +4.0723, for example, takes 7 bytes of memory. The bytes in memory are 2BH, 34H, 2EH, 30H, 37H, 32H, 33H where 2EH is the ascii representation of the decimal point, 32H is ascii for the number 2 etc. (see your ASCII CHART at the back of the small Programming Reference Guide). ASCII numbers are placed into memory in the AES-11 by transfer from the PC keyboard, the AES-11 keypad, by a BASIC11 POKE command, or you can place ascii numbers directly in your assembly program with the FCC directive. For example see the KEYCHAR routine in section II-B for using the keypad and see the program FLOAT.ASM on your AES-11 disk for using the FCC directive. (See the programs FLOAT, THERMO, and 12AD on your disk for examples of using floating point numbers in assembly programs.)

The ten bytes of RAM (**at addresses 6100H-6109H**) are used for the two 5-byte software floating point accumulators, FPACC1 in $6100-$6104 and FPACC2 in $6105-$6109. Each five-byte accumulator consists of a one-byte exponent, a three-byte mantissa, and one byte that is used to indicate the mantissa sign. The exponent byte is used to indicate the position of the binary point and is biased by decimal 128 ($80) to make floating-point comparisons easier. This one-byte exponent gives a dynamic range of about $1 \times 10^{\pm 38}$. The mantissa consists of three bytes (24 bits) and is used to hold both the integer and fractional portion of the floating-point number. The mantissa is always assumed to be "normalized" (i.e., most-significant bit of the most-significant byte a one). A 24-bit mantissa will provide slightly more than seven decimal digits of precision. A separate byte is used to indicate the sign of the mantissa rather than keeping it in twos complement form so that unsigned arithmetic operations may be used for manipulation of the mantissa. A positive mantissa is indicated by this byte being equal to zero ($00). A negative mantissa is indicated by this byte being equal to minus one ($FF). For example, the number pi and how it looks in floating-point format.

```
FPACC1 is  82 C9 0F DB 00  for number +3.1415927
FPACC2 is  82 C9 0F DB FF  for number -3.1415927
```

ERRORS
There are seven errors codes that may be returned by the floating-point package. When an error occurs, the condition is indicated to the calling program by setting the carry bit in the condition code register and returning an error code in the A-accumulator. Then a RTS instruction is issued and the next instruction in the calling program (your program) will be executed. You must test the carry bit upon returning from a floating-point subroutine if you want to use the error messages. You may do this for debugging a program.

Error #	Meaning
1	Format Error in ASCII to Floating-Point Conversion
2	Floating-Point Overflow
3	Floating-Point Underflow
4	Division by Zero (0)
5	Floating-Point Number too Large or Small to Convert to Integer
6	Square Root of a Negative Number
7	Tangent of $\pi/2$

These following floating-point routines are to be used with assembly.

Function	**Address**	**Description**
ASCFLT	$9050	ASCII TO FLOAT CONVERSION
FLTMUL	$9053	FLOATING-POINT MULTIPLY
FLTADD	$9056	FLOATING-POINT ADD
FLTSUB	$9059	FLOATING-POINT SUBTRACT
FLTDIV	$905C	FLOATING-POINT DIVIDE
FLTASC	$905F	FLOATING-POINT TO ASCII CONVERSION
FLTCMP	$9062	FLOATING-POINT COMPARE
UINT2FLT	$9065	UNSIGNED INTEGER TO FLOATING POINT
SINT2FLT	$9068	SIGNED INTEGER TO FLOATING POINT

FLT2INT	$906B	FLOATING POINT TO INTEGER
TFR1TO2	$906E	TRANSFER FPACC1 TO FPACC2
FLTSQR	$9071	FLOATING POINT SQUARE ROOT
FLTSIN	$9074	FLOATING POINT SINE
FLTCOS	$9077	FLOATING POINT COSINE
FLTTAN	$907A	FLOATING POINT TANGENT
DEG2RAD	$907D	DEGREES TO RADIANS CONVERSION
RAD2DEG	$9080	RADIANS TO DEGRESS CONVERSION
GETPI	$9083	PI

HOW TO USE THE FLOATING-POINT ROUTINES - An Example

Example: Multiply 3BHex(=59decimal) by 2.77854decimal and then multiply the result by -64.7733decimal.

1) Convert the hex number 3B to floating point format by using the routine UINT2FLT. First put $003B (make it a 16 bit number) into memory at $6102-$6103. This is the lower 2 bytes of the mantissa in FPACC1. This is just one of the uses of the location FPACC1.

```
LDD  #$003B          ;put hex number in 16 bit D Register
STD   $6102          ;transfer to memory starting at $6102
```

2) Call routine UNIT2FLT to convert the hex integer located at $6102 into a floating-point format and leave the result in FPACC1.

```
JSR   UNIT2FLT       ;see definition of UNIT2FLT below
```

3) Move the number in FPACC1 into FPACC2 so we leave FPACC1 free for the next part of the problem.

```
JSR  TFR1TO2         ;subroutine transfers number into FPACC2
```

4) Put the starting address of the buffer (max buffer size 14 bytes) which contains the number 2.77854 into the X Register. Say the required 7 byte buffer starts at RAM location $0140 (note it could also be in ROM since we are only going to read this buffer). Use ASCFLT to convert this ascii to floating-point format and store in FPACC1. ASCFLT goes to where X is pointed, expects to see a ascii format number up to 14 bytes long, converts the number to floating-point format and puts the result in FPACC1. Note that after reading our 7 byte number the routine ASCFLT will keep on reading thinking it may be a longer number. Therefore, to indicate the end of our number we must always put some ascii character not used by ASCFLT to indicate the end. Put the null character $00 in the 8th byte of the buffer to stop the conversion.

```
LDX  #$0140          ;have X "point to" $0140
JSR  ASCFLT          ;puts float format for 2.77854 in FPACC1
```

4) Give the assembly command JSR FLTMUL to multiply the numbers in FPACC1 and FPACC2. The result will be placed in FPACC1.

5) Now use FLTASC to convert the float format result in FPACC1 back into ascii and place it into a RAM buffer. We could use the same buffer at $0140; that way we do not have to change X. Or we could load in a new value for X and use a different buffer. Let's put the ascii result in an output buffer at $0150.

```
LDX  #$0150          ;answer buffer for 3bH * 2.77854
JSR  FLTASC          ;result now in output ascii buffer
```

6) Finally we want to multiple the number in the buffer at $0150 by the number -64.7733decimal. First use ASCFLT to convert the number in $0150 to float format in FPACC1. Next transfer this number to FPACC2. Next put -64.7733 in a 9 byte buffer (the ninth byte is $00) and use ASCFLT to convert it to float in FPACC1. Next use FLTMUL to multiply these numbers and put result in FPACC1. Finally use FLTASC to convert float in FPACC1 to ascii and store in a buffer somewhere in RAM pointed to by your choice of X.

In a real problem the number 3BH might be from a temperature or RPM sensor on a motor and the number 2.77854 might be a keypad input and the number -64.7733 might be a conversion constant. You should now be able to read the assembly files on your AES-11 disk where floating-points are used and understand how they work.

1. **Function: ASCFLT**
   ASCII-TO-FLOATING-POINT CONVERSION
   Operation: ASCII(X) → FPACC1
   Input: X register points to ASCII string to convert.
   Output: FPACC1 contains the floating-point number.
   Error code: Floating-point format error may be returned.

2. **Function: FLTMUL**
   FLOATING-POINT MULTIPLY
   Assembly: JSR FLTMUL
   Operation: FPACC1 x FPACC2 → FPACC1
   Input: FPACC1 and FPACC2 contain the number to be multiplied.
   Output: FPACC1 contains the product of the two floating-point accumulators. FPACC2 remains unchanged.
   Error code: Overflow, Underflow.

3. **Function: FLTADD**
   FLOATING-POINT ADD
   Operation: FPACC1+FPACC2 → FPACC1
   Input: FPACC1 and FPACC2 contain the numbers to be added
   Output: FPACC1 contains the sum of the two numbers. FPACC2 remains unchanged.
   Error code: Overflow, Underflow.

4. **Function: FLTSUB**
FLOATING-POINT SUBTRACT
Operation: FPACC1 - FPACC2 ➡ FPACC1
Input: FPACC1 and FPACC2 contain the numbers to be subtracted.
Output: FPACC1 contains the difference of the two numbers. FPACC2 remains unchanged.
Error codes:Overflow, Underflow.

5. **Function: FLTDIV**
FLOATING-POINT DIVIDE
Operation: FPACC1÷FPACC2 ➡ FPACC1
Input: FPACC1 and FPACC2 contain the devisor and dividend respectively.
Output: FPACC1 contains the quotient. FPACC2 remains unchanged.
Error codes:Divide by zero, Overflow, Underflow.

6. **Function: FLTASC**
FLOATING-POINT-TO-ASCII CONVERSION
Operation: FPACC1 ➡ (X)
Input: FPACC1 contains the number to be converted to an ASCII string. The index register X points to a 14 byte string buffer.
Output: The buffer pointed to by the X index register contains an ASCII string that represents the number in FPACC1. The string is terminated with a zero (0) byte and the X register points to the start of the string.
Error codes:None.

7. **Function: FLTCMP**
FLOATING-POINT COMPARE
Operation: FPACC1 - FPACC2
Input: FPACC1 and FPACC2 contain the numbers to be compared.
Output: Condition codes are properly set so that all branch instructions may be used to alter program flow. FPACC1 and FPACC2 remain unchanged.
Error codes:None.

8. **Function: UINT2FLT**
UNSIGNED INTEGER TO FLOATING-POINT
Operation: (16-bit unsigned integer) ➡ FPACC1
Input: The lower 16-bits of the FPACC1 mantissa contain an unsigned 16-bit integer.
Output: FPACC1 contains the floating-point representation of the 16-bit unsigned integer.
Error codes:None

9. **Function: SINT2FLT**
SIGNED INTEGER TO FLOATING-POINT
Operation: (16-bit signed integer) → FPACC1
Input: The lower 16-bits of the FPACC1 mantissa contain a signed 16-bit integer.
Output: FPACC1 contains the floating-point representation of the 16-bit signed integer.
Error codes:None

10. **Function: FLT2INT**
FLOATING-POINT TO INTEGER
Operation: FPACC1 → (16-bit signed or unsigned integer)
Input: FPACC1 may contain a floating-point number in the range 65535<=FPACC1<=-32767.
Output: The lower 16-bits of the FPACC1 mantissa will contain a 16-bit singed or unsigned number.
Error codes:None

11. **Function: TFR1TO2**
TRANSFER FPACC1 TO FPACC2
Operation: FPACC1 → FPACC2
Input: FPACC1 contains a floating-point numbers.
Output: FPACC2 contains the same number as FPACC2.
Error codes:None

12. **Function: FLTSQR**
SQUARE ROOT
Operation: √(FPACC1) → FPACC1
Input: FPACC1 contains a valid floating-point number.
Output: FPACC1 contains the squared root of the original number. FPACC2 is unchanged.
Error codes:SQUARE ROOT ERROR is returned if the number in FPACC1 is negative and FPACC1 remains unchanged.

13. **Function: FLTSIN**
SINE
Operation: SIN(FPACC1) → FPACC1
Input: FPACC1 contains an angle in radians in the range $-2\pi<=FPACC1<=2\pi$
Output: FPACC1 contains the sine of FPACC1, and FPACC2 remains unchanged.
Error codes:None

14. **Function: COSINE**
COSINE
Operation: COS(FPACC1) → FPACC1
Input: FPACC1 remains an angle in radians in the range $-2\pi<=FPACC1<=2\pi$
Output: FPACC1 contains the cosine of FPACC1, and FPACC2 remains unchanged.
Error codes:None

15. **Function: TANGENT**
TANGENT
Operation:TAN(FPACC1) ➡ FPACC1
Input: FPACC1 contains an angle in radians in the range $-2\pi <= FPACC1 <= 2\pi$
Output: FPACC1 contains the tangent of the input angle, and FPACC2 remains unchanged.
Error codes:Returns largest legal number if tangent of $\pm \pi/2$ is attempted.

16. **Function: DEG2RAD**
DEGREES TO RADIANS-CONVERSION
Operation: FPACC1 x π ÷ 180 ➡ FPACC1
Input: Any valid floating-point number representing an angle in degrees.
Output: Input angles equivalent in radians.
Error codes:None

17. **Function: RAD2DEG**
RADIANS TO DEGREES CONVERSION
Operation: FPACC1 x 180 ÷ π ➡ FPACC1
Input: Any valid floating-point number representing an angle in radians.
Output: Input angles equivalent in degrees.
Error codes:Overflow, Underflow.

18. **Function: PI**
PI
Operation: π ➡ FPACC1
Input: None
Output: The value of π is returned in FPACC1.
Error codes:None

IV. Preserved Memories for the AES Firmware

User program should never over write the following ram locations:

0000-0100H - Pseudo Interrupt Vector (shared both by the DEBUGGER and BASIC11)

6000H-603FH - HC11 I/O Registers

6040H-60FFH - AES11 MONITOR VARIABLES

6100H-6109H - Floating Point Accumulator #1
Floating Point Accumulator #2

6E00H-6FFFH - DEBUGGER 's DATA and STACK SPACE

7000H-7FFFH - DEVICES ADDRESS