AN1064

Use of Stack Simplifies M68HC11 Programming

By Gordon Doughman

Introduction

The architectural extensions of the M6800 incorporated into the M68HC11 allow easy manipulation of data residing on the stack of the microcontroller unit (MCU).

The M68HC11 central processor unit (CPU) automatically uses the stack for these two purposes:

- Each time the CPU executes a branch-to-subroutine (BSR) or jump-to-subroutine (JSR) instruction, it pushes a return address onto the stack. This procedure allows the CPU to resume execution with the instruction following the BSR or JSR when the program returns from the subroutine.
- Second, just before the MCU executes an interrupt service routine, the CPU saves its register contents on the stack, allowing the registers to be restored when the CPU executes a return-from-interrupt (RTI) instruction at the end of the interrupt service routine.

Two additional uses of the M68HC11 stack discussed in this application note are the storage of local or temporary variable values and subroutine parameter passing.

Using the stack for local variables and parameter passing provides the assembly language programmer with the following benefits:

- First, since a routine allocates storage space for local variables and parameters upon entry and releases the storage upon exit, the same temporary memory space can be reused by program routines that run in succession. This reuse can result in a substantial savings in the total amount of RAM required by a program.
- Second, allocating a new set of local variables and parameters when entering a routine makes it both re-entrant and recursive. Routines that possess these two properties can make a programmer’s job much easier when debugging a program in a real-time, interrupt-driven environment.
- Third, placing local variables and parameters on the stack helps to promote modular programming. Because all temporary storage required by a routine is allocated and deallocated by the program module itself, it can be easily detached from the main program for reuse or replacement.
- The final major benefit of using the stack for local variables and parameters becomes apparent during the debugging process. Because a routine’s local variables and parameters exist only while it is executing, it is very unlikely that one routine will accidentally modify the local variables and parameters of another routine. Once the programmer has written and debugged a routine, time can be spent finding logical errors and/or problems associated with the interaction of the different routines in a program.

The goal of this application note is to help the assembly language programmer understand the following topics:

- Basic operation of the M68HC11 stack
- Concept of the local and global variables
- Subroutine parameter passing
- Use of the M68HC11 instruction set to support local variables and parameter passing

The source code for the examples and the macros described in this application note can be obtained from http://www.mot.com/pub/SPS/MCU/appnotes

M68HC11 Stack Operation

The M68HC11 supports a stack through the use of the CPU stack pointer (SP) register. The SP is a 16-bit register that points to an area of RAM used for stack storage. Because the SP is 16 bits wide, the stack can be located anywhere in the M68HC11 64-Kbyte address space. The SP contents are undefined at power-up and are normally initialized in the first few instructions of a program. Each time a byte is pushed onto the stack, the SP is automatically decremented. Therefore, the initial value loaded into the SP is usually the address of the last RAM location in a system. Thus, as more information is pushed onto the stack, the stack area grows downward (the SP points to lower addresses) in the memory map. The SP always contains the address of the next available location on the stack.

As previously mentioned, the stack on the M68HC11 is used automatically by the CPU hardware during subroutine calls/returns and during the servicing of interrupts. When a subroutine is called by a JSR or BSR instruction, the address of the instruction following the JSR or BSR is automatically pushed onto the stack.

Since the M68HC11 only has an 8-bit data bus, two separate push operations are performed by the CPU hardware. During the first push operation, the low-order eight bits (b7–b0) of the return address are placed on the stack. The second push operation places the high-order eight bits (b15–b8) of the return address on the stack at the next lower address in memory. Performing the operation in this order leaves the 16-bit return address on the stack in the order that all 16-bit numbers are stored in memory, with the high-order eight bits at the lower address.

After a JSR or BSR instruction, the stack appears as shown in **Figure 1**.

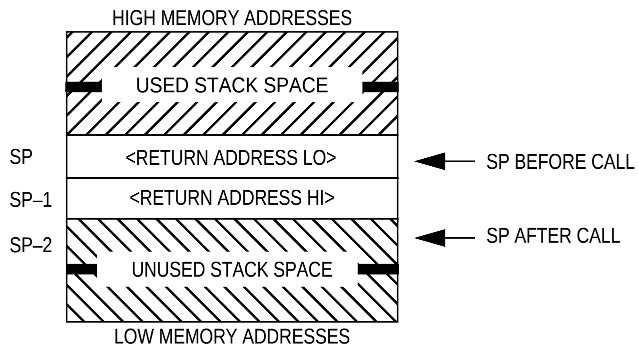


Figure 1. Stack Contents after Executing a JSR or BSR Instruction

Whenever an unmasked interrupt occurs, the contents of all CPU registers (with the exception of the SP itself) are pushed onto the stack as shown in **Figure 2**. After the registers are stacked, CPU execution continues at an address specified by the vector for the pending interrupt source. Upon completion of the interrupt service routine, the execution of an RTI instruction restores the previously saved CPU registers by pulling them off the stack in the reverse order in which they were pushed onto the stack. Since the entire state of the CPU is restored, execution resumes as if the interrupt had not occurred.

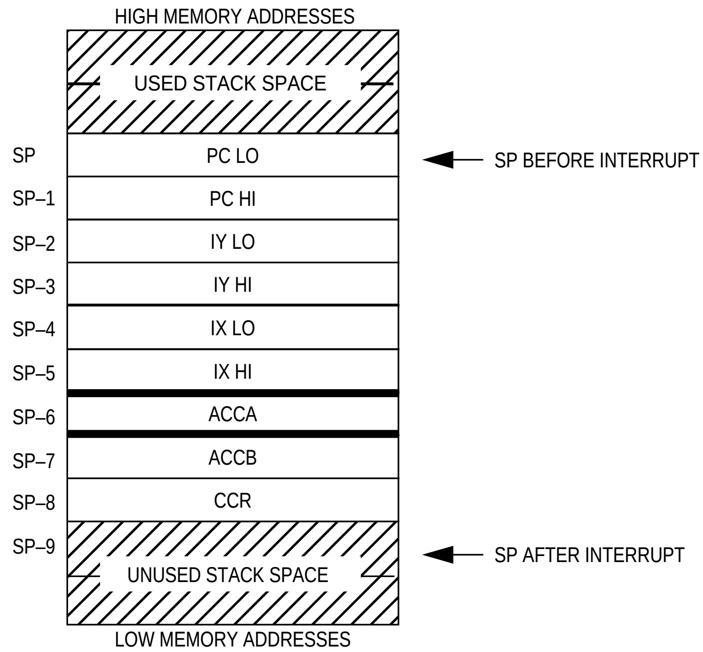


Figure 2. Stack Contents after an Interrupt

The M68HC11 instruction set contains instructions that allow the individual CPU registers to be pushed onto and pulled off the stack. For example, if the value contained in one of the CPU registers needs to be saved before a particular subroutine call, a push instruction places the register value on the stack. When the subroutine returns, a pull instruction restores the contents of the CPU register. These instructions not only allow the stack to be used as temporary data storage but also allow the construction of recursive and re-entrant subroutines.

M68HC11 instructions that involve the direct manipulation of the SP are listed in **Table 1**.

Table 1. Instructions Involving Direct Manipulation of the SP

Instruction Mnemonic	Description
PSHA	Push accumulator A onto the stack.
PSHB	Push accumulator B onto the stack.
PULA	Pull accumulator A off the stack.
PULB	Pull accumulator B off the stack.
PSHX	Push index register X onto the stack.
PSHY	Push index register Y onto the stack.
PULX	Pull index register X off the stack.
PULY	Pull index register Y off the stack.
INS	Increment the stack pointer by 1.
DES	Decrement the stack pointer by 1.
TXS	Place the contents of index register X – 1 in the stack pointer.
TYS	Place the contents of the index register Y – 1 in the stack pointer.
TSX	Place the contents of the stack pointer +1 in index register X.
TSY	Place the contents of the stack pointer in +1 in index register Y.

Stack Usage

Although most assembly language programmers use the M68HC11 stack for subroutine return addresses, register contents during interrupt processing and temporary CPU register storage, more powerful programming techniques can make additional use of the stack.

Most high-level language compilers for modern, block-structured, high-level languages make use of the stack for two additional functions: passing parameters and local or temporary variable storage. By borrowing some of these techniques, programmers can write assembly language programs that are much more reliable, easier to maintain, and easier to debug.

Variables in Assembly Language

Computer programs rarely operate on data directly; instead, the program refers to variables. A variable is a physical location in computer memory that can be used to hold different values while the program runs. Variables usually have an identifier or name associated with them. Using names to refer to data contained in memory is much easier than trying to remember a long string of binary or hexadecimal numbers.

Besides a name and an address, variables may have several other attributes. Depending on the programming language, variable declarations may assign attributes to the variables restricting both the scope and extent of the variable. The scope of a variable is the range of program text in which a particular variable is known and can be used. The extent of a variable is the time during which a computer associates physical storage with a variable name.

In assembly language, the scope of variables is usually global — for instance, variables may be referenced throughout the text of a program. Though some assemblers may provide mechanisms to restrict the scope of declared variables, many assembly language programmers do not use these features. A programmer using assembly language usually declares variables by employing an assembler directive as shown in **Listing 1**. This method assigns fixed storage locations to the variables. The extent of variables declared this way is for the entire program execution — for instance, the storage locations assigned to the variables at assembly time remain allocated during the entire time the program is executing.

```
*
*       RAM LOCATIONS
*
*
*
        ORG         $10
*
STANUM        RMB     1   STATION NUMBER REGISTER.
DATBLP        RMB     1   DATA TABLE POINTER REGISTER.
STAMSK        RMB     1   STATION BIT MASK REGISTER.
FCTNUM        RMB     1   FUNCTION NUMBER REGISTER FOR MODE SET.
XTEMP         RMB     2   X-REG. TEMPORARY STORAGE.
XTEMP1        RMB     2   X-REGISTER TEMPORARY STORAGE.
ATEMP1        RMB     1   A-REGISTER TEMPORARY STORAGE.
COUNT1        RMB     1   COUNT USED DURING STATION POLLING LOOP.
KPCNT         RMB     1   'NUMBER OF KEYS PESSED' COUNT.
LSTFCN        RMB     1   LAST T/L FUNCTION THAT WAS PROCESSED.
CALLST        RMB     1   REMOTE CALL STATUS BYTE.
ATEMP2        RMB     1   A-REG. TEMPORARY STORAGE FOR THE DELAY SUBROUTINE.
XTEMP3        RMB     2   X-REG. STORAGE BEFORE CALL TO DELAY SUBROUTINE.
COUNT2        RMB     1   COUNT USED IN DELAY SUBROUTINE.
NONESL        RMB     1   'NONE SELECTED' REGISTER USED BY SSCHK.
*
```

Listing 1. Declaring Global Variables in Assembly Language

Further examination of the variable declarations in **Listing 1** shows that several variables are used for intermediate calculation results or for temporary CPU register storage. This example is typical of the way many assembly language programmers allocate temporary storage. Each time they write a routine requiring temporary variable storage, they allocate an additional set of global variables. The use of this technique can lead to the inefficient use of RAM if there are many routines within a program requiring temporary storage.

In an effort to make more efficient use of the limited amount of RAM on single-chip MCUs, some programmers use a technique known as “variable sharing.” **Listing 2** shows a portion of a listing using this technique. In this program, more than one routine shares the use of a single temporary variable. To keep track of which routines use which variables, each line, in addition to the variable declaration, contains a list of the routines using that particular variable. In small programs, it may not be too difficult to manage temporary variables this way; however, in large programs having hundreds or thousands of routines using temporary variables, it becomes impossible to keep track of which routines use which temporary variables at any given time.

```
*
*       RAM LOCATIONS
*
*
*
        ORG             $0
*
***      variables - used by: ***
PTR0            RMB 2           main,readbuff,incbuff,AS
PTR1            RMB 2           main,BR,DU,MO,AS,EX
PTR2            RMB 2           EX,DU,MO,AS
PTR3            RMB 2           EX,HO,MO,AS
PTR4            RMB 2           EX,AS
PTR5            RMB 2           EX,AS,BOOT
PTR6            RMB 2           EX,AS,BOOT
PTR7            RMB 2           EX,AS
PTR8            RMB 2           AS
TMP1            RMB 1           main,hexbin,buffarg,termarg
TMP2            RMB 1           GO,HO,AS,LOAD
TMP3            RMB 1           AS,LOAD
TMP4            RMB 1           TR,HO,ME,AS,LOAD
```

Listing 2. Declaring Global Variables in Assembly Language

The sharing of temporary variable storage shown in **Listing 2** can produce debugging problems that are extremely hard to find. The chances of having one routine unintentionally modify the temporary storage of another can become quite high in large programs. In interrupt-driven, real-time systems, the sharing of temporary variables by various routines can become disastrous.

Consider the situation illustrated in **Figure 3**. Subroutine A and subroutine B both share the temporary variable Temp1. Initially, there seems to be no problem since subroutine A and subroutine B do not call one another. Yet, consider what happens if an interrupt occurs during the execution of subroutine A. Because of the interrupt, subroutine B is called indirectly through subroutine C. The execution of subroutine B causes any value placed in Temp1 by subroutine A before the interrupt to be overwritten! Because interrupts usually occur asynchronously to main program execution, the program may appear to operate properly most of the time and crash randomly, depending on when an interrupt occurs. This type of apparently random program failure can be almost impossible to find.

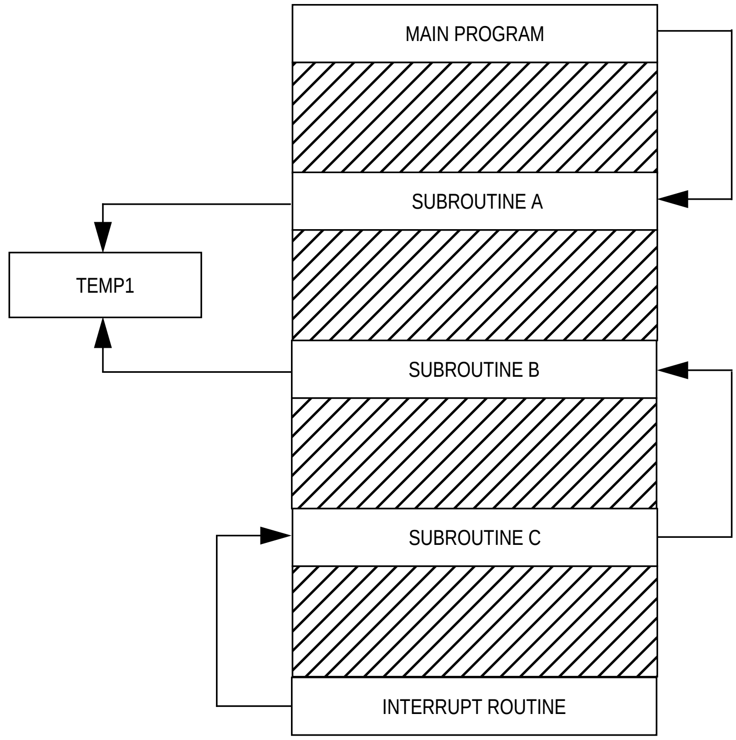


Figure 3. Two Subroutines Sharing a Single Temporary Variable

Although this example may seem overly simplistic, a program that contains hundreds or thousands of routines makes it nearly impossible to keep track of which subroutines are using what variables at any specific time, particularly if the main program and interrupt service routines share subroutines. The solution to this type of problem may seem simple — do not allow any subroutines to share globally declared temporary variables. This solution is acceptable provided enough RAM is available for all required temporary variables. A better solution to this problem can be found by examining the way modern, block-structured, high-level languages use temporary variables.

Variables in Block-Structured High-Level Languages

Most block-structured, high-level languages, notably C and Pascal, provide the ability to limit both the scope and the extent of variables as part of the language definition. In both C and Pascal, the scope of a variable is local to the block in which it is declared. The scope of variables declared outside of a block (function or procedure) is usually global. These global variables are similar to the ones declared in the assembly language shown in **Listing 1**. They can be accessed by all routines within a program, and they remain in existence throughout the entire time the program executes. **Listing 3** shows an example of how global variables are declared in C and Pascal.

```
        Pascal                                              C

var
  x,y:integer;                                        int x,y;
  j:char;                                             char j;
  z:boolean;                                          int z;
  num:array[1..10] of integer;                        int num[9];
  Date:record                                         struct Date {
    Month:integer;                                       int  x,y;
    Day:integer;                                         int Day;
    Year:integer;                                        int Year;
  end;  };

program(input,output);                                main()
.                                                     {
.                                                     .
.                                                     .
end.                                                  }
```

Listing 3. Declaring Global Variables in High-Level Languages

Variables declared within a function or procedure have their scope limited to that function or procedure. The extent of these variables is also limited. These variables, known as local or automatic variables, come into existence when the functions or procedures that contain them are called. When a function or procedure finishes execution, the local variables disappear, and the memory locations occupied by them can be used again. **Listing 4** shows an example of how local variables are declared in C and Pascal. In both examples, the variables `i` and `j` are local to procedure/function A and do not exist outside them.

Pascal | **C**

```
var
 x,y:integer;
 z:boolean;

procedure A;
  var
   i,j:integer;
  begin
.
.
.
end;
```

```
int x,y;
int z;

A()
{
 int i,j;
.
.
.
}
```

Listing 4. Declaring Local Variables in High-Level Languages

There are several benefits of using local variables:

- First, the restricted life of local variables can result in memory savings. Since storage for local variables is allocated upon entry to a routine and released upon exit from a routine, the same temporary memory space can be used by many different program routines. If two routines are run in succession, each can use the same storage locations.
- Second, since a new set of local variables is allocated each time the procedure or function is entered, it makes the routine both recursive and re-entrant. A re-entrant routine is one that allocates a new set of local variables upon entry. When complex programs are run in a real-time, interrupt-driven environment, the interrupt handlers may call the routine that was interrupted. Making routines re-entrant can greatly simplify a programmer's job during the debugging process in a real-time environment. The same properties that make a routine re-entrant also make a routine recursive. A recursive routine is one that can call itself.
- Third, the use of local variables helps to promote modular programming. A program module is a self-contained program element that can be easily detached from the main program either for reuse in another program or for replacement. Since any storage space for local variables is allocated and deal-located by the program module itself, the module code can easily be copied from a single place within one program and reused in another program.

- A fourth benefit of using local variables is evidenced during the debugging process. In complex programs, there may be hundreds or thousands of routines that have to interact with each other. Since local variables help isolate any changes made within a routine, debugging becomes a much simpler process. Once routines are written and debugged, the programmer does not have to worry about one routine accidentally modifying the local variables of another. Instead, time can be spent finding any logical errors and/or problems associated with the interaction of routines in the program.

Even with all the benefits provided by the use of local variables, there are some costs associated with their use. On the M68HC11, programs using local variables tend to be slightly larger and slower than programs using only global variables because the addressing modes required to access the local variables can make the instruction somewhat longer and may cause longer execution time. Given the benefits of using local variables, a slightly larger and slower program is usually well worth the cost.

The reusable memory storage for local variables is usually taken from the same memory space used for the MCU's hardware stack. Placing local variables on the hardware stack leaves them intact even if the routine using them is interrupted. The specifics of allocating, deallocating, and accessing local variables residing on the M68HC11 stack is discussed in **Using the M68HC11 Stack**.

Passing Parameters

To make routines more flexible and to vary their actions each time they are called, different information must be passed to the routines. Generally, most assembly language programmers use the CPU registers to pass information to a subroutine. Using this technique is acceptable as long as the amount of information to be passed to the subroutine fits within the available CPU registers.

When the amount of information to be passed to a routine exceeds the space available in the CPU registers, the information can be passed in a set of global variables. This technique may be acceptable for some situations, but it can also cause problems that make debugging difficult. One problem with passing parameters in this manner is that it makes a routine non-re-entrant. Referring to **Figure 4**, assume that subroutine A's parameters are passed in a set of global variables. If subroutine A is called either by the main program or by subroutine C as a result of an interrupt, the program will work correctly. If an interrupt occurs during the execution of subroutine A, the original parameters passed by the main program will be overwritten when subroutine C calls subroutine A. When the processor returns from the interrupt and resumes execution of subroutine A, it will be using incorrect parameter data, and the results passed back to the main program will most likely be incorrect.

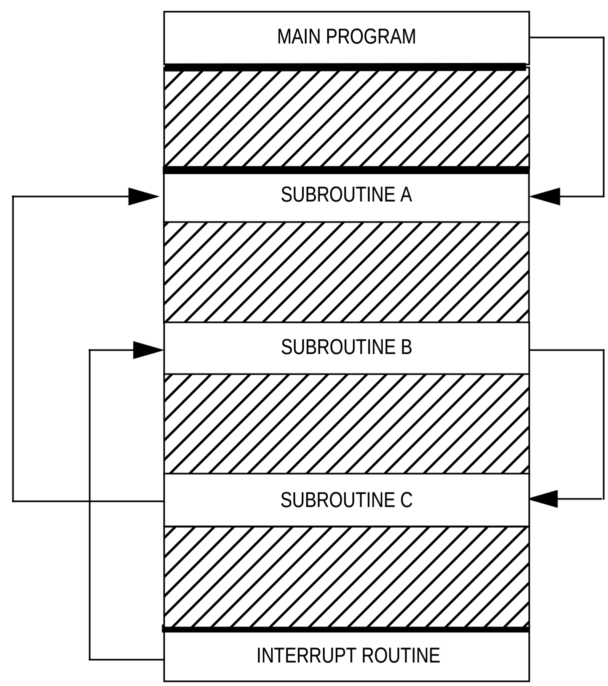


Figure 4. Subroutine Calling Chain

Because interrupts usually occur asynchronously to main program execution, the program may appear to operate properly most of the time and crash randomly. This type of problem can be extremely difficult to locate and can make debugging of real-time, interrupt-driven systems very difficult. Passing the parameters on the stack completely solves this problem. When subroutine C calls subroutine A as a result of the interrupt, a new set of parameters is placed on the stack while the original parameters remain undisturbed. **Figure 5** shows the state of the stack after an interrupt.

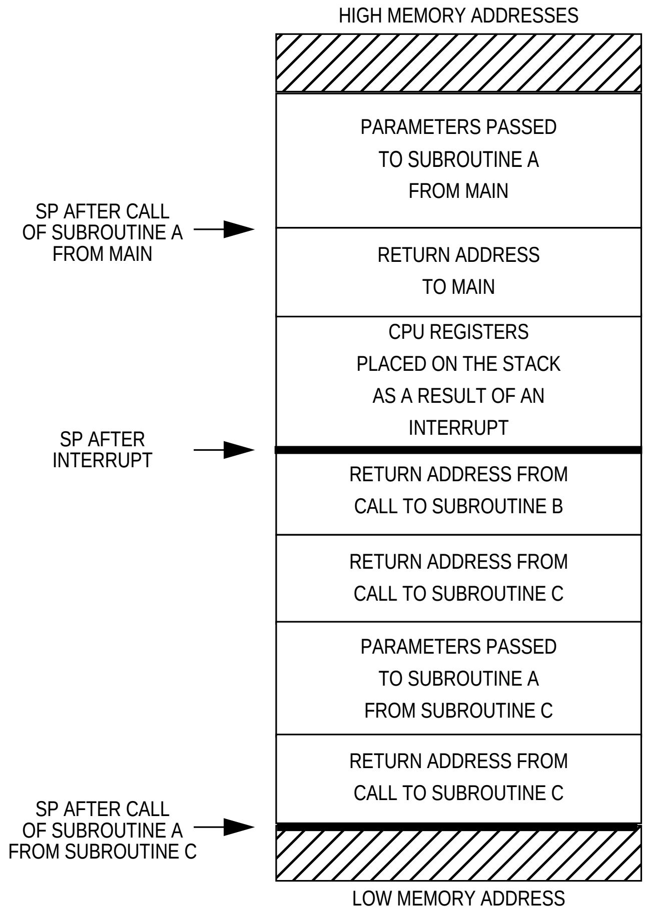


Figure 5. Stack State as a Result of an Interrupt

In addition to where parameters are passed, there is also an issue of how parameters are passed. Subroutine parameters can be passed either by value or by reference. When a parameter is passed by value, the parameter acts as a local variable whose initial value is provided by the calling routine. Any modification of the supplied value has no effect on the original data that was passed to the subroutine. Thus a subroutine can import values but not export values by means of value parameters.

Passing a parameter by reference is one method used to pass results back to a calling subroutine. These types of parameters are known as variable parameters. When using variable parameters, the address of the actual parameter is passed to the subroutine rather than a value. The passed address can be a local variable of the calling routine or even the address of a global variable. Whenever a subroutine has to effect a permanent change in the values passed to it, the parameters must be passed by reference rather than by value.

Consider the following example in both C and Pascal that exchanges the value of two integers:

```
              Pascal                                          C
           Call By Value                                 Call By Value

procedure SwapInt (x,y:integer);          void SwapInt (int x,y)
  var                                       {
    Temp:integer;                           int Temp;
    begin                                   Temp=x;
      Temp:=x;                              x=y
      x:=y                                  y=Temp
      Y:=Temp                               }
  end;

          Call By Reference                          Call By Reference

procedure SwapInt (var x,y:integer);      void SwapInt (int *x, *y)
  var                                       {
    Temp:integer;                           int Temp;
    begin                                   Temp=*x;
      Temp:=x;                              *x=*y
      x:=y                                  *y=Temp
      y:=Temp                               }
  end;

Call Of "SwapInt" Using Either Method     Call Of "SwapInt" Using Call by Reference

program(output);                          main( )
  var                                       {
  z,w:integer;                              int w,z;
    begin                                   z=2;
      z:=2;                                 w=4;
      w:=4;                                 SwapInt (&z,&w);
      SwapInt (z,w);                       }
  end;
```

Listing 5. Passing Parameters by Reference and by Value

If the call-by-value routine were to be used in this example, the routine would not work as the programmer might expect. It would exchange the local values of `x` and `y` within the `SwapInt` routine, but it would have no effect on the actual variables in the routine's call statement. For the `SwapInt` routine to work properly, the routine must be declared so that the parameters are passed by reference rather than by value. As mentioned previously, passing a parameter by reference passes the address of the actual parameter. In the example in **Listing 5**, using the call-by-reference routine, the addresses of the variables `z` and `w` are passed to the `SwapInt` routine when it is called from the main program. This procedure allows the `SwapInt` routine to exchange the actual values of the variables passed to the routine.

Function/ Subroutine Return Values

Most subroutines or functions, if they are to perform a useful action in a program, will return one or more values to the calling routine. Any value or status can be returned using one of the three methods previously described. When a subroutine only needs to return a single value, one of the CPU registers is commonly used to pass the value back to the calling routine. This simple, safe technique allows the routine to remain re-entrant. This method is used most often by C compilers to return a value from a function.

Similar to the situation that exists when passing parameters in the CPU registers, there may be times when a routine must return more information than will fit in the CPU registers. The information can be returned in a set of global variables; however, as previously described, this method poses the same problems as passing parameters in this manner. Returning results in global variables makes the routine non-re-entrant and can cause the same debugging problems previously described.

A better way to return large amounts of data from a subroutine is to allocate the required amount of space on the stack either just before or just after pushing a routine's parameters onto the stack. This method possesses the same benefits of passing parameters on the stack — it makes the routine completely re-entrant and self-contained. Most Pascal compilers return function values in this manner.

Using the M68HC11 Stack

This section specifically discusses how to allocate, deallocate, and access both local variables and parameters residing on the M68HC11 stack. The programmer's model of the M68HC11 is shown in **Figure 6**. The following paragraphs briefly describe the CPU registers and their usage.

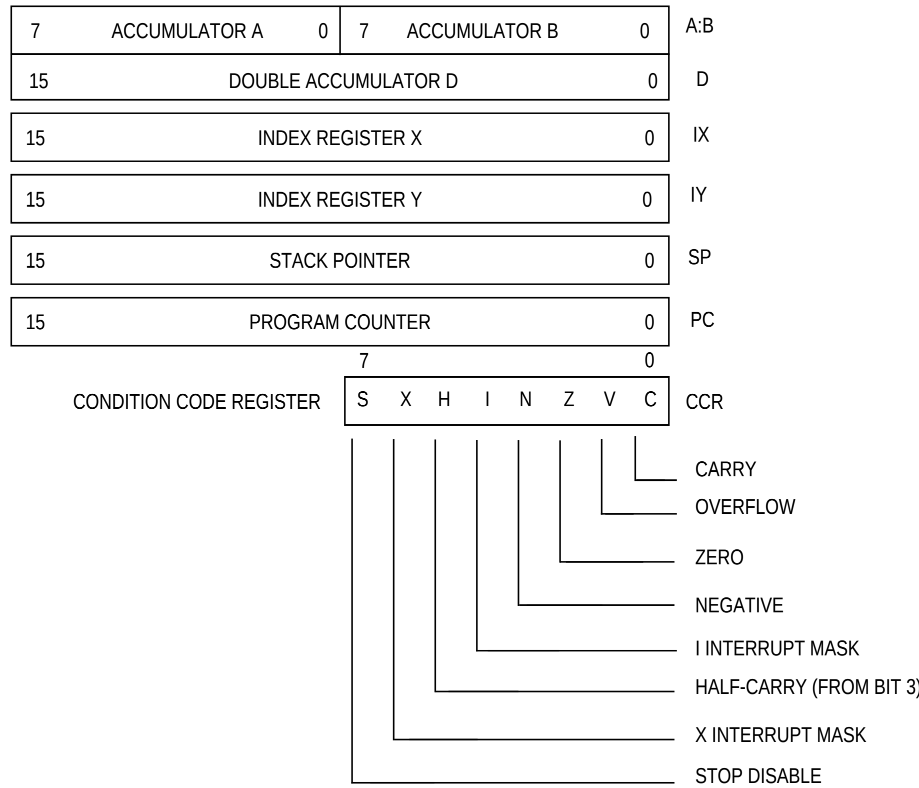


Figure 6. M68HC11 Programmer's Model

The A and B accumulators are used to hold operands and the results of arithmetic and logic operations. These two 8-bit registers can be concatenated to form a single 16-bit D accumulator to support the M68HC11 16-bit arithmetic instructions. The A and B accumulators can easily be used to push data onto or pull data off the stack.

The X and Y index registers are used in conjunction with the CPU indexed addressing mode. The indexed addressing mode uses the contents of the 16-bit index register in addition to a fixed 8-bit unsigned offset that is part of the instruction to form the effective address of the operand to be used by the instruction. The index registers play a very important role in accessing data residing on the stack.

The CPU SP is a 16-bit register that points to an area of RAM used for stack storage. The stack is used automatically during subroutine calls to save the address of the instruction that follows the call. When an interrupt occurs, the stack is used automatically by the CPU to save the entire CPU register contents on the stack (except for the SP itself). The SP always contains the address of the next available location on the stack.

The program counter (PC) is a 16-bit register used to hold the address of the next instruction to be executed.

The condition code register (CCR) contains five status indicators and two interrupt mask bits. The status bits reflect the results of arithmetic and other operations of the CPU as it performs instructions.

Before considering the specifics of parameter passing and the utilization of local variables that reside on the M68HC11 stack, the method used to access the information placed on the stack will be discussed. One M68HC11 index register and the CPU indexed addressing mode are used to access parameters or local variables residing on the stack. With respect to the indexed addressing mode, the contents of one of the 16 bit index registers plus a fixed unsigned offset is used in calculating the effective address of an instruction's operand. The unsigned offset, contained in a single byte following the instruction opcode, can only accommodate positive offsets in the range 0–255. Thus, the indexed addressing mode can only access information at addresses that are between 0 and 255 bytes greater than the base address contained in one of the index registers. Figure 7 illustrates how to calculate the effective address of an instruction using the indexed addressing mode.

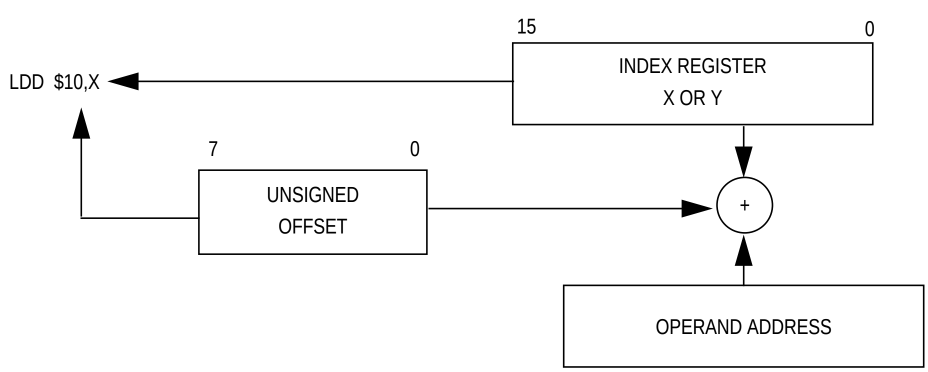


Figure 7. Effective Address Calculation for Indexed Addressing Mode

As information is pushed onto the M68HC11 stack, the SP is decremented, signifying that the information placed on the stack resides at addresses greater than the address contained in the SP. The use of indexed addressing is ideal for accessing information residing on the M68HC11 stack. The example shown in **Figure 8** illustrates how information on the stack is manipulated.

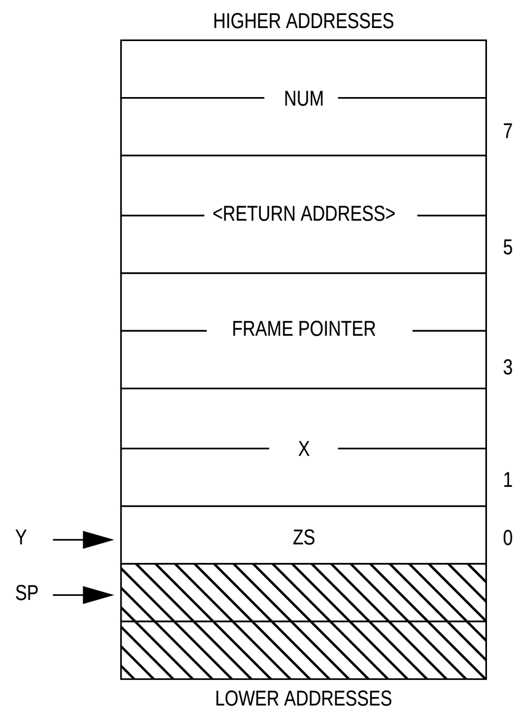


Figure 8. Stack Data Access Example

As **Figure 8** shows, the SP is pointing to the next available address, and the Y index register is pointing to the last data placed on the stack. The instruction `LDD 1,Y` will load the value of the local variable `x` into the D accumulator. To access the parameter `Num`, the instruction `LDD 7,Y` can be used. Any instructions that support the indexed addressing mode can be used to manipulate stack data.

Passing Parameters

Parameters are easily placed on the M68HC11 stack by CPU push instructions. **Table 2** lists the push instructions available on the M68HC11. Note that there is not a single instruction for pushing the D accumulator onto the stack. A PSHD instruction can easily be simulated by executing the two instructions PSHB and PSHA. These two instructions must be executed in this order to keep the value pushed onto the stack consistent with the way 16-bit values are stored in memory — for example, 16-bit values are placed in memory with the most significant eight bits at a lower address than the least significant eight bits. By following this convention, a 16-bit parameter pushed onto the stack in this manner is easily retrieved using one of the 16-bit load instructions.

Table 2. Push Instructions in the M68HC11 Instruction Set

Instruction Mnemonic	Description
PSHA	Push accumulator A onto the stack.
PSHB	Push accumulator B onto the stack.
PSHX	Push index register X onto the stack.
PSHY	Push index register Y onto the stack.

As previously mentioned, parameters can be passed either by value or by reference. Consider a function, `Int2Asc`, that converts a signed 16-bit integer to ASCII text and places the ASCII characters in a text buffer.

The function requires two parameters: the number to be converted into ASCII text and a pointer to a buffer where the ASCII text is to be stored. The first parameter is passed to the subroutine by value because the actual number to be converted is passed to the function. The second parameter is passed by reference because a pointer to the buffer is passed to the routine and not the buffer itself.

A function declaration written in C is shown in **Listing 6**.

```
void Int2Asc(int Num; char *Buff)
     {
       int Pwr10 = 10000;
       char zs = 0;
       .
       .
       .
       }
```

Listing 6. Function Declaration of Int2Asc

Before calling an equivalent routine written in M68HC11 assembly language, the two parameters will be pushed onto the stack as shown in **Listing 7**.

```
LDX     ErrorNum        ; Get the value of the current error.
PSHX                    ; Place it on the stack.
LDX     #OutBuff        ; Get the address of the Output buffer.
PSHX                    ; Place it on the stack.
JSR     Int2Asc         ; Go convert the number.
```

Listing 7. Placing Parameters on the M68HC11 Stack

Using the immediate addressing mode with the second load index register X (LDX) instruction loads the address of OutBuff into the X index register rather than the 16-bit value contained in the memory locations OutBuff and OutBuff+1. After both parameters have been pushed onto the stack, the function is called with a JSR instruction. Upon entry to the subroutine Int2Asc, the parameters reside just above the return address, as shown in **Figure 9**.

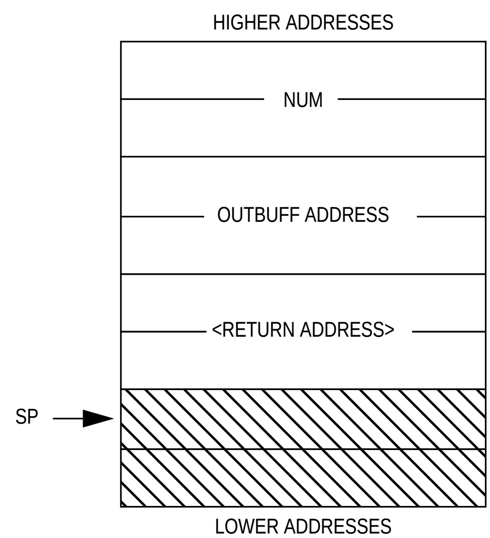


Figure 9. Location of Parameters Passed on the Stack

Allocating Local Variables

Four basic techniques can be used to allocate local variables that reside on the stack. Choosing which one to use depends upon the total amount of storage required for the local variables and whether the variables need to have an initial value assigned to them. Of course, a combination of all four techniques can be used.

One technique used to allocate space on the stack for local storage involves the use of the decrement stack pointer (DES) instruction. The DES instruction subtracts one from the value of the SP each time the instruction is executed, allocating one byte of local variable storage for each DES instruction. This technique is a simple and direct way of allocating local storage but becomes impractical when large amounts of local storage are required. For instance, if 100 bytes of local storage are required for a subroutine, 100 DES instructions are needed to allocate the required amount of storage. This required amount is clearly unacceptable since each DES instruction requires one byte of program memory. Even if a small program loop is set up to execute 100 DES instructions, the subroutine will suffer a severe execution speed penalty each time the routine is entered.

Using the previously described technique requires one byte of program storage for each byte of local storage that is allocated. Since allocating local storage simply involves decrementing the SP, the PSHX instruction can be used to allocate two bytes of local storage space for each executed PSHX instruction. The actual contents of the X index register are irrelevant because the only concern is decrementing the SP. The use of this technique can be confusing if not properly documented, since it is not directly obvious what is being accomplished with five or six sequentially executed PSHX instructions.

Many times it is necessary to initialize local variables with a particular value before they are used. The same technique used to push parameters onto the stack before a subroutine call also can be used to allocate space for local variables and simultaneously assign initial values to them. This procedure is accomplished by loading one of the CPU registers with a variable's initial value and executing a PSH instruction. The program fragment in **Listing 8** shows the use of this technique to allocate and initialize both an 8- and 16-bit local variable.

```
Int2Asc      equ     *
             .
             .
             ldx     #10000     ; get the initial value of Pwr10.
             pshx               ; allocate and initialize it.
             clra               ; initial value of zs is zero.
             psha               ; allocate and initialize it.
```

Listing 8. Allocating and Initializing Local Variables

If more than 13 bytes of local storage are required by a subroutine, a fourth technique allocates storage more efficiently than using multiple DES or PSHX instructions. Since there are not any instructions that allow arithmetic to be performed directly on the SP, the fourth technique involves using several M68HC11 instructions. These instructions adjust the value of the SP downward in memory, allocating the required amount of local storage. Listing 9 shows the instruction sequence required to allocate an arbitrary number of bytes of local storage.

```
SinCos     equ     *
           .
           .
           tsx                ; SP+1 → X.
           xgdx               ; exchange the contents of x and d.
           subd    #xxxx      ; subtract the required amt. of storage.
           xgdx               ; place the result back into x.
           txs                ; X–1 → SP. Update the SP.
```

Listing 9. Allocation of More Than 13 Bytes for Local Storage

Since no single instruction allows the contents of the SP to be transferred to the D accumulator, the 2-instruction sequence transfer from SP to index register X or Y; exchange double accumulator and index register X or Y (TSX; XGDX, or TSY; XGDY) must be used. Placing the SP value in the D accumulator allows the use of the 16-bit subtract instruction to adjust the value of the SP. The subtract double accumulator (SUBD) instruction will subtract the 16-bit value xxxx from the contents of the D accumulator. To place this new value in the SP, the 2-instruction sequence XGDX; TXS or XGDY; TYS is used.

NOTE: *Actually, the TSX or TSY instruction causes the SP value plus 1 to be transferred to either the X or Y index register (SP + 1 → X or SP + 1 → Y). This transfer does not pose a problem because when the SP is updated with the TXS or TYS instruction 1 is subtracted from the value of the index register (X – 1 → SP or Y – 1 → SP) before the SP is updated. Remember that since the SP points to the next available location on the stack, adding 1 to its value before the execution of the TSX or TSY instruction makes the X or Y index register point to the last data placed on the stack.*

Creating a Complete Stack Frame

In addition to providing storage space for local variables and parameters, a complete stack frame (sometimes called an activation record) must contain two additional pieces of information: a return address and a pointer to the base of the stack frame of any previous routines. The return address is placed on the stack automatically by the M68HC11 when it executes either a JSR or BSR instruction. As shown in **Figure 9**, the return address is placed on the stack just below a subroutine's parameters.

Before using either the X or Y index register to access a routine's parameters or local variables, the contents of the register must first be saved. The index register contents, known as the stack frame pointer, may contain the base address of a stack frame for a routine from which control was transferred. This pointer must be maintained so that when control is returned to the calling routine, the calling routine's environment can be restored to its previous state. Even if a routine has no local variables or parameters, the contents of the index register being used as the stack frame pointer must be saved before the register is used for any other purpose.

The best time to save the value of the previous stack frame pointer is immediately upon entry to a subroutine, which places the previous stack frame pointer immediately below the return address (see **Figure 10**).

After space for local variables has been allocated, the stack frame pointer for the new subroutine needs to be initialized. By transferring the contents of the SP to either the X or Y index register using the TSX or TSY instruction, a new stack frame is created.

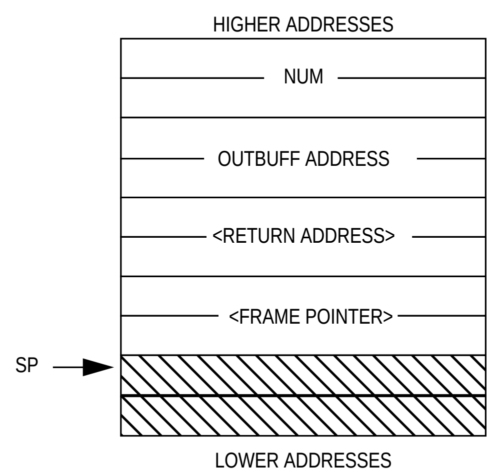


Figure 10. Location of the Stack Frame Pointer

In summary, creating a complete stack frame involves the following three steps after entering a subroutine:

1. Immediately upon entry to a subroutine, the contents of the index register being used as the stack frame pointer must be saved by using either the PSHX or PSHY instruction.
2. Storage space for the routine's local variables should be allocated using one of the three methods described earlier.
3. The new stack frame pointer must be initialized using either the TSX or TSY instruction.

The last issue to discuss is which index register to use as the stack frame pointer. In terms of code size and speed, the X index register would be the most logical choice since all instructions involving the Y index register require one additional opcode byte and one additional clock cycle to execute. However, if a program is not making extensive use of the stack for local variables and parameters but is performing extensive array or table manipulations, the Y index register may be a better choice. No matter which index register is used as the stack frame pointer, it should be, if at all possible, dedicated to that use throughout a program. Program debugging is much easier if the contents of a single index register can always be expected to point to the current stack frame.

Accessing Parameters and Local Variables

As mentioned in **Using the M68HC11 Stack**, local variables and parameters are accessed by using instructions that support the indexed addressing mode. The following list identifies the local and store instructions as well as all arithmetic and logic instructions that support indexed addressing. Because most M68HC11 instructions support indexed addressing, it is just as code efficient to manipulate local variables that reside on the stack as it is to manipulate global variables using direct or extended addressing. **Figure 11(a)** illustrates a complete allocation frame as used by a subroutine.

ADCA	ADCB	ADDA	ADDB	ADDD
ANDA	ANDB	ASL	ASR	BCLR
BITA	BITB	BRCLR	BRSET	BSET
CLR	CMPA	CMPB	COM	CPD
CPX	CPY	DEC	EORA	EORB
INC	JMP	JSR	LDAA	LDAB
LDD	LDS	LDX	LDY	LSL
LSR	NEG	ORA	ORB	ROL
ROR	SBCA	SBCB	STAA	STAB
STD	STS	STX	STY	SUBA
SUBB	SUBD	TST		

Using the indexed addressing mode to access data contained in a stack frame places a restriction on the combined size of local variables and parameters. Since the indexed addressing mode functions by adding an unsigned 8-bit offset to the contents of the 16-bit index register, the indexed addressing mode can only access information at addresses that are between 0 and 255 bytes greater than the base address contained in one of the index registers. Consequently, the maximum size of a single stack frame is restricted to 256 bytes. If no parameters are passed to a routine on the stack, then the entire 256 bytes are available for local variables. However, when parameters are passed on the stack, not only is the space occupied by the parameters unavailable for use as local variables, but the subroutine return address and previous stack frame pointer reduce the amount of available space by an additional four bytes.

In most embedded control applications that use the M68HC11 in single-chip mode, this limit on the combined size of parameters and local variables for a single stack frame is rarely a concern since the amount of on-chip RAM is limited. Several techniques can be used to work around the limit imposed by the indexed addressing mode; however, they are extremely wasteful in terms of code space and execution speed.

NOTE: *In reality, the amount of memory available for local storage in a single stack frame is 257 bytes. Because the M68HC11 is capable of loading and storing 16 bits of data with a single instruction, it is possible to access one byte beyond the contents of the index register plus the fixed offset of 255 with the 16-bit load and store instructions.*

Deallocating the Stack Frame

When a subroutine has completed execution, the stack space allocated for the stack frame must be released so the memory can be reused by subsequent subroutine calls. The deallocation of the stack frame includes not only the removal of the space occupied by the local storage, but also the restoration of the previous stack frame pointer and the removal of space occupied by any parameters that were passed to the subroutine.

The process of freeing the memory occupied by the stack frame is simply a matter of adjusting the value of the SP upward in memory. The

SP must be adjusted upward by the same amount that it was adjusted downward when the space for the stack frame was allocated. Either of the following methods can be used to perform this task.

The most obvious way to perform the deallocation is to reverse the process used to allocate the storage. Removing the stack frame in this manner involves these three basic steps.

First, the storage occupied by any local variables must be removed from the stack area by using the reverse of one of the techniques described in **Allocating Local Variables**. Alternately, the technique shown in **Listing 10** can be used. This technique involves adjusting the value of the SP upward in memory by the same amount it was adjusted downward when the space was allocated.

```
LBAD    #LOCLEN     Get size of local storage into the B register.
ABX                 Add it to the current stack frame pointer.
TXS                 Deallocate the local storage.
```

Listing 10. Alternate Method for Deallocating Local Storage

Second, the previous stack frame pointer must be restored. Because the previous stack frame pointer is now on the top of the stack, the use of a pull index register X or Y from the stack (PULX or PULY) instruction is all that is needed to perform this operation. At this point, the return address is on the top of the stack. Simply executing a return-from-subroutine (RTS) instruction returns program execution to the instruction following the subroutine call.

After returning to the calling routine, any parameters that were pushed onto the stack before the subroutine call must now be removed. This places the burden of removing subroutine parameters on the calling routine rather than on the called routine. This method of removing subroutine parameters is perfectly acceptable and is the one most often used by C language compilers.

Removing the parameters can be as simple as a 1-instruction operation. If the X or Y index register contains the address of the current stack frame pointer, simply executing a TXS or TYS instruction places the SP just below the stack frame pointer. If the X or Y index register does not contain the address of the current stack frame pointer, an alternate method must be used to remove the parameters. **Figure 11** illustrates the state of the stack at each stage of the deallocation process.

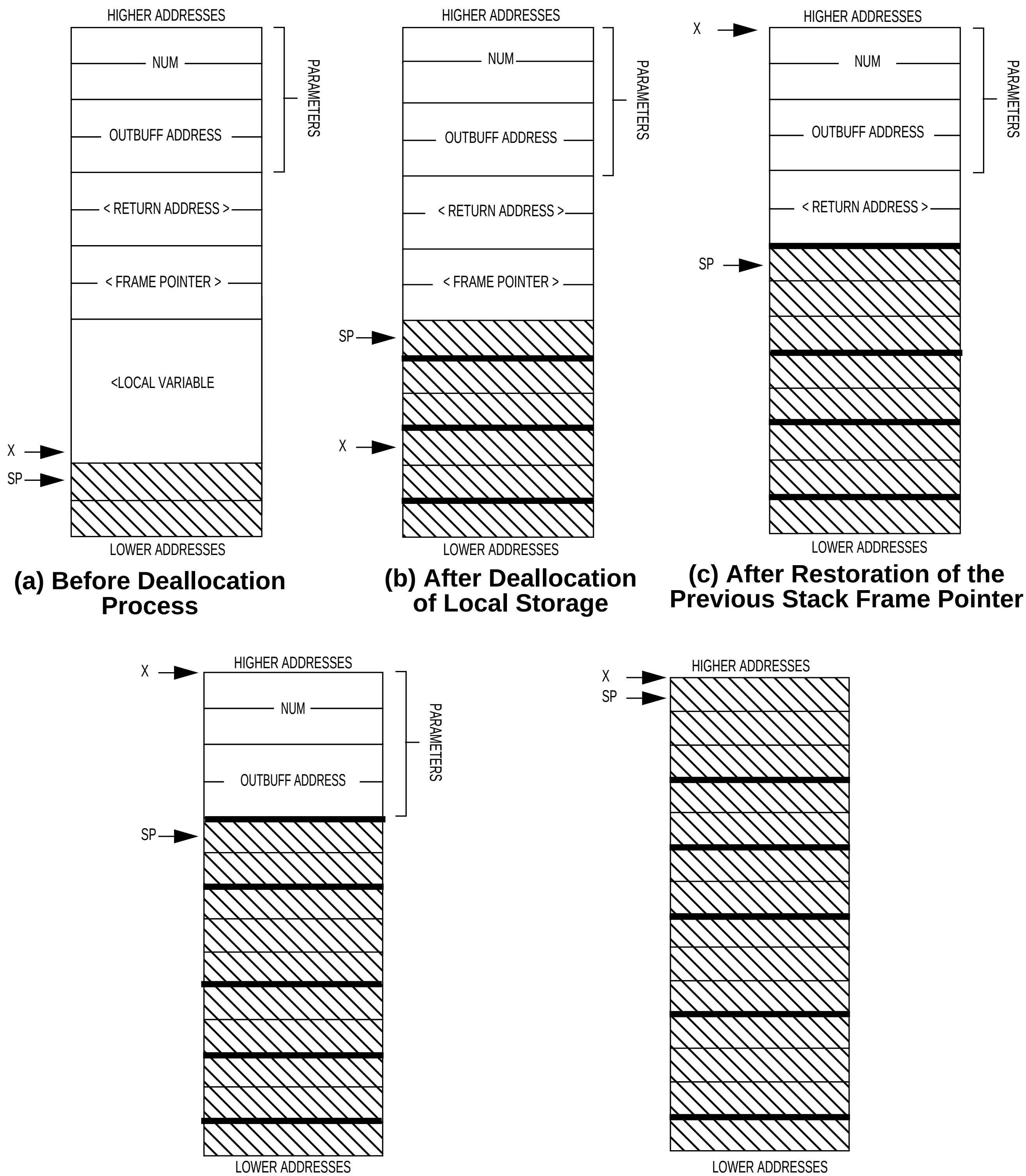
An alternative method requires the called routine to remove the entire stack frame, including any parameters passed to it. This method may not be as code efficient as the first method since it requires a fixed number of instructions to release the storage space occupied by the entire stack frame. **Listing 11** shows the instruction sequence necessary to deallocate the stack frame when the X index register is being used as the stack frame pointer. This 4-instruction sequence requires nine bytes of program storage space and 18 cycles to execute but removes the entire stack frame, regardless of the size. This method of stack frame deallocation has one drawback — the X or Y index register must always contain a valid stack frame pointer. Thus, all subroutines, even if they do not require parameters or local variables, must “mark” the current state of the stack upon entry by executing a PSHX; TSX or PSHY; TSY instruction sequence.

NOTE: *In **Listing 11**, RA is the offset value to the <Return Address> and PSFP is the offset value to the <Previous Stack Frame Pointer>.*

```
LDY     RA,X        Load the return address into the Y register.
LDX     PSFP,X      Restore the previous stack frame pointer.
TXS                 Remove the entire stack frame.
JMP     0,Y         Return to the calling routine.
```

Listing 11. Alternate Method for Deallocating Entire Stack Frame

In summary, choosing a method to deallocate the stack frame involves a trade-off between code size and execution speed. Using the first method results in the smallest amount of code being generated but may take longer to execute than the method shown in **Listing 11**.



(a) Before Deallocation Process

(b) After Deallocation of Local Storage

(c) After Restoration of the Previous Stack Frame Pointer

(d) After Execution of an RTS Instruction

(e) After Deallocation of Parameters

Figure 11. Deallocation of the Stack Frame

Support Macros

The following macros may be used to help in managing stack frames in M68HC11 programs. Using these macros may not provide the smallest or fastest code in all situations but should make the program easier to write and debug. Although the macros were written for the Micro Dialects µASM-HC11™ assembler that runs on the Macintosh®, they can be used with other assemblers with some modification. The following paragraph explains the way parameters are passed and referenced in the Micro Dialects assembler and should help in the conversion process.

When a macro is defined, parameters are not declared. When a macro is invoked, the parameters appear in the operand field following the macro name. Within a macro definition, parameters are referenced by using a colon (;) followed by a single decimal digit (0–9). Therefore, within the body of the macro, the first parameter is referenced by using :0, the second parameter is referenced by using :1, and so forth. Parameter substitution is performed strictly on a textual substitution basis.

The link macro shown in **Listing 12** can be used to allocate a complete stack frame after entry into a subroutine. The link macro performs the following functions:

- Saves the previous stack frame pointer
- Allocates the required number of bytes of local storage
- Initializes a new stack frame pointer

The calling convention for the link macro is:

```
link        <s.f. reg>,<storage bytes>
```

The first parameter passed to the macro is the name of the index register being used as the stack frame pointer (either X or Y). Although no check is made to ensure that a legal index register name is passed to the macro, the assembler will produce an "Unrecognized Mnemonic" error message when the macro is expanded. The second parameter is the number of bytes of local storage required by the subroutine.

µASM-HC11 is a trademark of Micro Dialects.
Macintosh is a registered trademark of Apple Computer, Inc.

```
link    macro
        psh:0        ; Save the previous stack frame pointer.
        ts:0         ; Transfer the stack pointer into :0.
        xgd:0        ; Transfer :0 into D.
        subd  #:1    ; subtract the required amount of local storage.
        xgd:0        ; Initialize the new stack frame pointer.
        t:0s         ; Update the stack pointer with new value.
        endm
```

Listing 12. Link Macro

The return and deallocate (rtd) macro shown in **Listing 13** can be used to partially deallocate a subroutine stack frame. The rtd macro performs the following functions:

- Deallocates local storage
- Restores the previous stack frame pointer
- Returns to the calling routine

The rtd macro does not remove any parameters from the stack that may have been passed to the subroutine. Removal of any parameters must be performed by the calling routine. This macro is useful when no parameters are passed to a subroutine or when parameters are passed in registers. The calling convention for the rtd macro is:

```
rtd   <s.f. reg>,<storage bytes>
```

Like the link macro, the first parameter passed to the rtd macro is the name of the index register being used as the stack frame pointer (either X or Y). Again, although no check is made to ensure that a legal index register name is passed to the macro, the assembler will produce an "Unrecognized Mnemonic" error message when the macro is expanded. The second parameter is the number of bytes of local storage allocated when the subroutine was entered.

```
rtd    macro
       ldab  #:1    ; number of bytes to deallocate.
       ab:0         ; add it to the current stack frame pointer.
       t:0s         ; deallocate storage by updating the stack pointer.
       pul:0        ; restore the previous stack frame pointer.
       rts          ; return to the calling routine.
       endm
```

Listing 13. Return and Deallocate Macro

The only drawback in using this macro is that it uses the B accumulator when deallocating a subroutine's local storage, preventing a subroutine from returning a 16-bit result in the D accumulator. A simple solution to the problem is to surround the load accumulator B (LDAB) and add accumulator B to index register X or Y (ABX/ABY) instructions with the PSHB/PULB instruction pair as shown in **Listing 14**. This macro, renamed frtd for function return and deallocate, allows the D accumulator to be loaded with a return value immediately before the macro is called. A second solution to this problem is to place all return values on the stack as described in **Function/ Subroutine Return Values**, allowing the calling routine to retrieve the returned value and then remove it along with the parameters.

```
frtd    macro
        pshb         ; save the lower byte of the return value.
        ldab  #:1    ; number of bytes to deallocate.
        ab:0         ; add it to the current stack frame pointer.
        pulb         ; restore the lower byte of the return value.
        t:0s         ; deallocate storage by updating the stack pointer.
        pul:0        ; restore the previous stack frame pointer.
        rts          ; return to the calling routine.
        endm
```

Listing 14. Function Return and Deallocate Macro

The return and deallocate using x (rtdx) and return and deallocate using y (rtdy) macros shown in **Listing 15** can be used to completely deallocate a subroutine stack frame, including any parameters that were passed on the stack. The rtdx and rtdy macros perform the following functions:

- Deallocates the entire stack frame, including local storage and passed parameters
- Restores the previous stack frame pointer
- Returns to the calling routine

The calling convention for the rtdx and rtdy macros is as follows:

```
rtdx    <storage bytes>    or rtdy    <storage bytes>
```

The only parameter passed to the macros is the number of local storage bytes allocated upon entry to the subroutine. These macros have an advantage over the rtd macro in that the A and B accumulators are not used during deallocation, which allows a return value to be loaded into the A, B, or D registers before execution of the rtdx or rtdy macro.

```
rtdx   macro
       ldy     :0+2,x   ; Load the return address into the Y index register.
       ldx     :0,x     ; restore the previous stack frame pointer.
       txs              ; Update the stack pointer, removing the storage space.
       jmp     0,y      ; Return to the calling routine.
       endm
*
*
rtdy   macro
       ldx     :0+2,y   ; Load the return address into the X index register.
       ldy     :0,y     ; restore the previous stack frame pointer.
       tys              ; Update the stack pointer, removing the storage space.
       jmp     0,x      ; Return to the calling routine.
       endm
```

Listing 15. rtdx and rtdy Macros

The only restriction to using the rtdx and rtdy macros is that a valid stack frame pointer for the previous subroutine must be present in either the X or Y index register when the register is pushed onto the stack at the beginning of the subroutine. Even if a subroutine has no local variables in it or no parameters passed to it, a PSHX and TSX instruction must be executed immediately upon entry to a subroutine to save the previous stack frame pointer and "mark" the current state of the stack. Before returning, a PULX instruction must be executed to restore the previous stack frame pointer.

This restriction implies that, somewhere in the program, the index register to be used as the stack frame pointer must be initialized with a valid value. If either the X or Y index register is to be dedicated for use as a stack frame pointer, the index register must be initialized at the beginning of the program. The initial value loaded into the index register should be one more than the value loaded into the stack pointer, which is easily accomplished by executing the TSX instruction immediately after initializing the stack pointer.

In summary, the use of the rtdx and rtdy macros are convenient in that they remove both parameters and local variables passed to subroutines. However, their use will cost three extra instructions in subroutines that do not have local variables or parameters but call subroutines that use local variables or have parameters passed to them.

Examples

Appendix A. Example Listings contains several examples that use the techniques described to manage local storage, parameter passing, and allocation/deallocation of stack frames.

Appendix A. Example Listings

```
                                        Include "Stack Macros"
          ******************************************************************************************
          *
          *                                        Written By
          *                                      Gordon Doughman
          *
          *
          *
          *             The author reserves the right to make changes to this file. Although this
          *             software has been carefully reviewed and is believed to be reliable, neither
          *             Freescale nor the author assumes any liability arising from its use. This soft-
          *             ware may be freely used and/or modified at no cost or obligation to the user.
          *
          *             The following macros may be used to help in managing stack frames in
          *             M68HC11 programs. The macros were written for Micro Dialects µASM-HC11
          *             assembler that runs on the Macintosh but may be used with other assemblers
          *             with some modification. The following discussion of the way parameters are
          *             passed and referenced should help in the conversion process.
          *
          *             Within a macro, parameters are referenced by using a colon (:) followed
          *             by a single decimal digit (0–9). Therefore, within the body of the macro
          *             the first parameter is referenced by using ’:0’, the second parameter is
          *             referenced by using ’:1’, and so forth. Parameter substitution is performed
          *             strictly on a textual substitution basis.
          *
          ******************************************************************************************
          *
          *             The link macro may be used to allocate a complete stack frame after entry
          *             into a subroutine. The link macro performs the following functions:
          *             1) Saves the previous stack frame pointer; 2) Allocates the requested
          *             number of bytes of local storage; 3) Initializes a new stack frame pointer.
          *
          *             Usage: link <s.f. reg>,<storage bytes>
          *
          *             The first parameter passed to link is the index register that is being used
          *             as the stack frame pointer (either x or y). Although no check is made to
          *             ensure that a legal index register name is passed to the macro, the assembler
          *             will produce an "Unrecognized Mnemonic" error message when the macro is
          *             expanded. The second parameter is the number of bytes of local storage
          *             required by the subroutine.
          *
          ******************************************************************************************
          *
M         link          macro
M                       psh:0                      ; Save the previous stack frame pointer.
M                       ts:0                       ; Transfer the stack pointer into :0.
M                       xgd:0                      ; Transfer :0 into D.
M                       subd  #:1                  ; subtract the required amount of local storage.
M                       xgd:0                      ; Initialize the new stack frame pointer.
M                       t:0s                       ; Update the stack pointer with new value.
M                       endm
          *
          *
```

```
            ****************************************************************************************
            *
            *               The rtd (Return and Deallocate) macro may be used to partially deallocate
            *               a subroutine stack frame that includes parameters passed on the stack. The
            *               rtd macro performs the following functions: 1) Deallocates local storage;
            *               2) Restores the previous stack frame pointer; 3) Returns to the calling
            *               routine. Rtd DOES NOT remove any parameters from the stack. This function
            *               must be performed by the calling routine. This macro is useful when
            *               parameters are passed in registers rather than on the stack.
            *
            *               Usage: rtd   <s.f. reg>,<storage bytes>
            *
            *               The first parameter passed to link is the index register that is being used
            *               as the stack frame pointer (either x or y). Although no check is made to
            *               ensure that a legal index register name is passed to the macro, the assembler
            *               will produce an "Unrecognized Mnemonic" error message when the macro is
            *               expanded. The second parameter is the number of bytes of local storage
            *               used by the subroutine.
            *
            ****************************************************************************************
            *
M           rtd             macro
M                           ldab  #:1                ; number of bytes to deallocate.
M                           ab:0                     ; add it to the current stack frame pointer.
M                           t:0s                     ; deallocate storage by updating the stack pointer
M                           pul:0                    ; restore the previous stack frame pointer
M                           rts                      ; return to the calling routine
M                           endm
            *
            *
            ****************************************************************************************
            *
            *               The frtd (Function Return and Deallocate) macro may be used to partially
            *               deallocate a subroutine stack frame that includes parameters passed on
            *               the stack. The frtd macro performs the following functions: 1) Deallocates
            *               local storage; 2) Restores the previous stack frame pointer; 3) Returns
            *               to the calling routine. Frtd DOES NOT remove any parameters from the stack.
            *               This function must be performed by the calling routine. This macro is
            *               useful when parameters are passed in registers rather than on the stack and
            *               a value is being returned in the D-accumulator.
            *
            *               Usage: frtd  <s.f. reg>,<storage bytes>
            *
            *               The first parameter passed to frtd is the index register that is being used
            *               as the stack frame pointer (either x or y). Although no check is made to
            *               ensure that a legal index register name is passed to the macro, the assembler
            *               will produce an "Unrecognized Mnemonic" error message when the macro is
            *               expanded. The second parameter is the number of bytes of local storage
            *               used by the subroutine.
            *
            ****************************************************************************************
            *
M           frtd            macro
M                           pshb                     ; save the lower byte of the return value.
M                           ldab  #:1                ; number of bytes to deallocate.
M                           ab:0                     ; add it to the current stack frame pointer.
M                           pulb                     ; restore the lower byte of the return value.
M                           t:0s                     ; deallocate storage by updating the stack pointer
M                           pul:0                    ; restore the previous stack frame pointer.
M                           rts                      ; return to the calling routine.
M                           endm
```

```
        *
        *
        *******************************************************************************************
        *
        *               The rtdx and rtdy (Return and Deallocate using x or y) macros may be used
        *               to completely deallocate a subroutine stack frame including parameters that
        *               were passed on the stack. The rtdx macro performs the following functions:
        *               1) Deallocates the entire stack frame including local storage and passed
        *               parameters; 2) restores the previous stack frame pointer; and 3) Returns
        *               to the calling routine.
        *
        *               Usage: rtdx  <storage bytes>
        *               Usage: rtdy  <storage bytes>
        *
        *               The only parameter passed to the routines is the number of bytes of local
        *               storage that were originally allocated upon entry to the subroutine. These
        *               macros have the advantage that the a and b accumulators are not used during the
        *               deallocation process. This allows a value to be loaded into a, b, or d register
        *               before execution of the rtdx or rtdy macro and returned to calling routine.
        *
        *******************************************************************************************
        *
M       rtdx            macro
M                       ldy           :0+2,x        ; Load return address into the y-index register.
M                       ldx           :0,x          ; restore the previous stack frame pointer
M                       txs                         ; Update stack pointer, removing storage space.
M                       jmp            0,y          ; Return to the calling routine.
M                       endm
        *
        *
M       rtdy            macro
M                       ldx           :0+2,y        ; Load return address into the x-index register.
M                       ldy           :0,y          ; restore the previous stack frame pointer.
M                       tys                         ; Update stack pointer, removing storage space.
M                       jmp            0,x          ; Return to the calling routine.
M                       endm
        *
        *
        *******************************************************************************************
        *
        *               The pshd macro pushes the 16-bit d-accumulator onto the stack. The
        *               b-accumulator is pushed first so that the least significant 8-bits of
        *               the 16-bit number appear on the stack at the higher address. This is
        *               consistent with the way all 16-bit numbers are stored in memory.
        *
        *               Usage:  pshd
        *
        *               No parameters are required by the macro.
        *
        *******************************************************************************************
        *
M       pshd            macro
M                       pshb
M                       psha
M                       endm
        *
        *
```

```
      **************************************************************************************
      *
      *               The puld macro pulls the top two bytes from the stack and places them in
      *               the 16-bit d-accumulator. The first byte pulled from the stack is placed
      *               in the a-accumulator; the second byte pulled from the stack is placed in
      *               the b-accumulator. The pull order is consistent with the way all 16-bit
      *               numbers are stored in memory.
      *
      *               Usage:    puld
      *
      *               No parameters are required by the macro.
      *
      **************************************************************************************
      *
M     puld            macro
M                     pula
M                     pulb
M                     endm
      *
      *
      **************************************************************************************
      *
      *               The clrd macro uses the clra and clrb instructions to clear the 16-bit
      *               d-accumulator.
      *
      *               Usage:    clrd
      *
      *               No parameters are required by the macro.
      *
      **************************************************************************************
      *
M     clrd            macro
M                     clra
M                     clrb
M                     endm
      *
      *
      *
      **************************************************************************************
      *
      *                                              Written By
      *                                            Gordon Doughman
      *
      *
      *
      *               The author reserves the right to makes changes to this file. Although this
      *               software has been carefully reviewed and is believed to be reliable, neither
      *               Freescale nor the author assumes any liability arising from its use. This soft-
      *               ware may be freely used and/or modified at no cost or obligation to the user.
      *
      *
      *
      **************************************************************************************
      *
      *               This subroutine converts a 16-bit binary integer to a null terminated
      *               ASCII string. Three parameters are passed to the subroutine on the
      *               stack. The first parameter is the 16-bit binary number to be converted.
      *               The second parameter is the address of a buffer where the null terminated
      *               ASCII string will be placed. The buffer should be at least 7 bytes long.
      *               The third parameter is a boolean flag indicating whether the number passed
      *               in the first parameter is a signed or unsigned 16-bit number. If the byte
```

```
                  *               flag is zero, the number is converted as an unsigned number. If the byte
                  *               is non-zero, the number will be converted as a 16-bit signed number.
                  *               Parameters are pushed onto the stack in the following order 1) Signed Flag;
                  *               2) Pointer to ASCII buffer; 3) Number to be converted. A typical
                  *               calling sequence would be:
                  *
                  *               clra                        ; Do the conversion as an unsigned number
                  *               psha                        ; put the flag on the stack.
                  *               ldd          #Buffer        ; get the address of the ascii buffer.
                  *               pshd                        ; put the address on the stack.
                  *               ldd          Num            ; Get the number to convert.
                  *               pshd                        ; Put it on the stack
                  *               jsr          Int2Asc        ; Go convert the number.
                  *               .
                  *               .
                  *               .
                  *
                  *               This subroutine has two local variables. The first, zs, is a boolean variable
                  *               used to suppress leading zeros when doing a conversion. It is located at an
                  *               offset of 0 from the stack frame pointer. The second local, Divisor, is a 16-
                  *               bit variable. It is used to divide the number being converted by succeedingly
                  *               lower powers of 10. Divisor is located at an offset of 1 from the local stack
                  *               frame pointer.
                  *
                  *               NOTE: This routine was written assuming that the previous stack frame pointer
                  *               is the x-index register. HOWEVER, because the x-index register is required
                  *               by the integer divide instruction, the y-index register is used as the
                  *               stack frame pointer WITHIN the Int2Asc subroutine.
                  *
                  *
                  *    Declare locals
                  *
0000              PCSave          set          *              ; save the current PC value
0000                              org          0              ; set PC to 0 for offsets to locals
0000              zs              rmb          1              ; declare zs variable.
0001              Divisor         rmb          2              ; declare Divisor variable.
0003              LocSize         set          *              ; number of bytes of local storage.
0000                              org          PCSave
                  *
                  * Offsets to parameters
                  *
0007              Num    equ      LocSize+4                   ; offset to Num parameter.
0009              BuffP           equ          LocSize+6      ; offset to BuffP parameter.
000B              Signed          equ          LocSize+8      ; offset to Signed parameter.
                  *
0000              Int2Asc         equ          *
0000 3C           [ 4]            pshx                        ; save the previous stack frame pointer.
0001 CC2710       [ 3]            ldd          #10000         ; initialize the divisor to 10000.
0004                              pshd
0004 37           [ 3]            pshb
0005 36           [ 3]            psha
0006 4F           [ 2]            clra                        ; initialize zs to 0.
0007 36           [ 3]            psha
0008 1830         [ 4]            tsy                         ; initialize the new stack frame pointer.
000A 18EC07       [ 6]            ldd          num,y          ; get the number to convert. Is it zero?
000D 260B         [ 3]            bne          Int2Asc1       ; no go do the conversion.
000F CC3000       [ 3]            ldd          #$3000         ; yes.
0012 CDEE09       [ 6]            ldx          BuffP,y        ; point to the buffer.
0015 18ED00       [ 6]            std          0,y            ; just put an ASCII 0 in the buffer.
0018 2050         [ 3]            bra          Int2Asc5       ; then return.
001A 186D0B       [ 7] Int2Asc1   tst          Signed,y       ; do the conversion as a signed number?
```

```
001D 2716     [ 3]          beq       Int2Asc2     ; no.
001F 4D       [ 2]          tsta                   ; yes. Is the number negative?
0020 2A13     [ 3]          bpl       Int2Asc2     ; no. just go do the conversion.
0022 43       [ 2]          coma                   ; yes. make it a positive number by negation.
0023 53       [ 2]          comb
0024 C30001   [ 4]          addd      #$1
0027 18ED07   [ 6]          std       Num,y        ; save the result
002A 862D     [ 2}          ldaa      #'-'         ; get an ASCII minus sign.
002C CDEE09   [ 6]          ldx       BuffP,y      ; point to the buffer.
002F A700     [ 4]          staa      0,x          ; put it in the buffer
0031 08       [ 3]          inx                    ; point to the next location in the buffer.
0032 CDEF09   [ 6]          stx       BuffP,y      ; save the new pointer value.
0035 18EC07   [ 6] Int2Asc2 ldd       Num,y        ; get the remainder to convert.
0038 CDEE01   [ 6]          ldx       Divisor,y
003B 02       [41]          idiv
003C 18ED07   [ 6]          std       Num,y        ; save the remainder.
003F 8F       [ 3]          xgdx                   ; put the dividend into d.
0040 5D       [ 2]          tstb                   ; was the dividend 0?
0041 2605     [ 3]          bne       Int2Asc3     ; no. go store the number in the buffer
0043 186D00   [ 7]          tst       zs,y         ; are we still suppressing leading zeros?
0046 2710     [ 3]          beq       Int2Asc4     ; yes. go setup for the next divide.
0048 CB30     [ 2] Int2Asc3 addb      #'0'         ; make the dividend ASCII.
004A 8601     [ 2]          ldaa      #1
004C 18A700   [ 5]          staa      zs,y         ; don't suppress leading zeros anymore.
004F CDEE09   [ 6]          ldx       BuffP,y      ; get a pointer to the buffer.
0052 E700     [ 4]          stab      0,x          ; save the digit.
0054 08       [ 3]          inx                    ; point to the next location.
0055 CDEF09   [ 6]          stx       BuffP,y      ; save the new pointer value.
0058 18EC01   [ 6] Int2Asc4 ldd       Divisor,y    ; get the previous divisor.
005B CE000A   [ 3]          ldx       #10
005E 02       [41]          idiv                   ; divide it by 10.
005F CDEF01   [ 6]          stx       Divisor,y    ; save the dividend. Is it zero?:
0062 26D1     [ 3]          bne       Int2Asc2     ; no. continue with the conversion.
0064 CDEE09   [ 6]          ldx       BuffP,y      ; get a pointer to the buffer.
0067 6F00     [ 6]          clr       0,x          ; null terminate the string.
0069 30       [ 3]          tsx                    ; this is only needed because we are using y as
                                                     our sf pointer.
006A              Int2Asc5  rtdx      LocSize      ; return & deallocate locals & parameters
006A 1AEE05   [ 6]          ldy       LocSize+2,x  ; Load the return address into y-index register.
006D EE03     [ 5]          ldx       LocSize,x    ; restore the previous stack frame pointer.
006F 35       [ 3]          txs                    ; Update stack pointer, removing storage space.
0070 186E00   [ 4]          jmp       0,y          ; Return to the calling routine.
              *
              *
              *
              ***************************************************************************************
              *
              *             This subroutine performs a 16 x 16 bit unsigned multiply and produces a 32-bit
              *             result. Two 16-bit numbers are passed to the subroutine on the stack.
              *             The 32-bit result is returned on the stack in place of the two 16-bit
              *             parameters. This allows the calling routine to easily pull the product
              *             from the stack and store the result. Because multiplication is a
              *             commutative operation, the order in which the parameters are pushed
              *             onto the stack is unimportant. A typical calling sequence would be:
              *
              *             ldd       Num1
              *             pshd
              *             ldd       Num2
              *             pshd
              *             jsr       Mul16x16
              *             puld
```

```
                *               std         Result32
                *               puld
                *               std         Result32+2
                *   .
                *   .
                *   .
                *
                *               This subroutine has four local variables. Each variable occupies 1 byte
                *               on the stack. These four bytes are used to hold the partial product as
                *               the final answer is being computed. These four byte variables are
                *               treated as 16-bit variables during the calculation.
                *
                *               NOTE: This routine was written assuming that the stack frame pointer
                *               is the x-index register.
                *
                *       Declare locals
                *
0073            PCSave          set         *             ; save the current PC value
0000            *               org         0             ; set PC to 0 for offsets to locals
0000            Prd0            rmb         1             ; declare ms byte of partial product variable.
0001            Prd1            rmb         1             ; declare next ms byte of partial product variable
0002            Prd2            rmb         1             ; declare next ls byte of partial product variable
0003            Prd3            rmb         1             ; declare ls byte of partial product variable.
0004            LocSize         set         *             ; number of bytes of local storage.
0073                            org         PCSave
                *
                *      Offsets to parameters
                *
0008            Fact1           equ         LocSize+4     ; offset to factor 1 parameter.
000A            Fact2           equ         LocSize+6     ; offset to factor 2 parameter.
                *
                        cycles clear
                *
0073            Mul16x16        equ         *
0073 3C         [ 4]            pshx                      ; save the previous stack frame pointer.
0074                            clrd                      ; clear the d-accumulator.
0074 4F         [ 2]            clra
0075 5F         [ 2]            clrb
0076                            pshd                      ; allocate & initialize the locals prd0 - prd3
0076 37         [ 3]            pshb
0077 36         [ 3]            psha
0078                            pshd
0078 37         [ 3]            pshb
0079 36         [ 3]            psha
007A 30         [ 3]            tsx                       ; initialize the new stack frame pointer.
007B A609       [ 4]            ldaa        Fact1+1,x     ; get the ls byte of factor 1.
007D E60B       [ 4]            ldab        Fact2+1,x     ; get the ls byte of factor 2.
007F 3D         [10]            mul                       ; multiply them.
0080 ED02       [ 5]            std         Prd2,x        ; save the first term of the partial product.
0082 A608       [ 4]            ldaa        Fact1,x       ; get the ms byte of factor 1.
0084 E60B       [ 4]            ldab        Fact2+1,x     ; get the ls byte of factor 2.
0086 3D         [10]            mul                       ; multiply them.
0087 E301       [ 6]            addd        Prd1,x        ; add the result into the partial product.
0089 ED01       [ 5]            std         Prd1,x        ; save the result.
008B A609       [ 4]            ldaa        Fact1+1,x     ; get the ls byte of factor 1.
008D E60A       [ 4]            ldab        Fact2,x       ; get the ms byte of factor 2.
008F 3D         [10]            mul                       ; multiply them.
0090 E301       [ 6]            addd        Prd1,x        ; add the result into the partial product.
0092 ED01       [ 5]            std         Prd1,x        ; save the result.
0094 2402       [ 3]            bcc         Mul16         ; Was there a carry into Prd0?
0096 6C00       [ 6]            inc         Prd0,x        ; yes. 'add' it in.
```

```
0098 A608      [ 4] Mul16    ldaa      Fact1,x       ; get the ms byte of factor 1.
009A E60A      [ 4]          ldab      Fact2,x       ; get the ms byte of factor 2.
009C 3D        [10]          mul                     ; multiply them.
009D E300      [ 6]          addd      Prd0,x        ; add it to the partial product.
009F ED08      [ 5]          std       Fact1,x       ; overwrite the two parameters with the result.
00A1 EC02      [ 5]          ldd       Prd2,x
00A3 ED0A      [ 5]          std       Fact2,x
00A5                         rtd       x,LocSize     ; return and deallocate the locals.
00A5 C604      [ 2]          ldab      #LocSize      ; number of bytes to deallocate.
00A7 3A        [ 3]          abx                     ; add it to the current stack frame pointer.
00A8 35        [ 3]          txs                     ; deallocate storage by updating stack pointer.
00A9 38        [ 5]          pulx                    ; restore the previous stack frame pointer.
00AA 39        [ 5]          rts                     ; return to the calling routine.
                      cycles total=170               ; Total number of E cycles for a 16 x 16 multiply.
               *
               *
               ******************************************************************************************
               *
               *         This subroutine performs a 32 by 16 bit unsigned divide and produces a 32-bit
               *         quotient and a 16-bit remainder. Both the divisor and dividend are passed to
               *         the subroutine on the stack. The 32-bit quotient and 16-bit remainder are
               *         returned on the stack in place of the divisor and dividend. This allows the
               *         calling routine to easily pull the answer from the stack and store the result.
               *         The divisor is pushed onto the stack first, followed by the lower 16-bits of
               *         the dividend and finally the upper 16-bits of the dividend. A typical calling
               *         sequence would be:
               *
               *         ldd       Divisor
               *         pshd
               *         ldd       Dividend+2
               *         pshd
               *         ldd       Dividend
               *         pshd
               *         jsr       Div32x16
               *         puld
               *         std       Quotient
               *         puld
               *         std       Quotient+2
               *         puld
               *         std       Remainder
               *         .
               *         .
               *         .
               *
               *
               *         This subroutine has two local variables. A 32-bit variable for partial
               *         quotient results that is treated as two 16-bit variables and a 16-bit
               *         variable for intermediate remainder results.
               *
               *         NOTE: This routine was written assuming that the previous stack frame pointer
               *         is the x-index register. HOWEVER, because the x-index register is required
               *         by the integer and fractional divide instructions, the y-index register is
               *         used as the stack frame pointer WITHIN the Div32x16 subroutine.
               *
               *      Declare locals
               *
00AB           PCSave        set       *             ; save the current PC value.
0000                         org       0             ; set PC to 0 for offsets to locals.
0000           Quo0          rmb       2             ; declare upper 16-bits of quotient.
0002           Quo2          rmb       2             ; declare lower 16-bits of quotient.
0004           Rem           rmb       2             ; declare remainder.
```

```
0006           LocSize        set          *              ; number of bytes of local storage.
00AB                          org          PCSave
               *
               *       Offsets to parameters
        *
000A           Num0           equ          LocSize+4      ; upper 16-bits of Dividend.
000C           Num2           equ          LocSize+6      ; lower 16-bits of Dividend.
000E           Denm           equ          LocSize+8      ; 16-bit divisor.
               *
                       cycles clear
               *
00AB           Div32x16       equ          *
00AB 3C        [ 4]           pshx                        ; save the previous stack frame pointer.
00AC                          clrd                        ; clear the d-accumulator
00AC 4F        [ 2]           clra
00AD 5F        [ 2]           clrb
00AE                          pshd                        ; allocate & initialize the locals.
00AE 37        [ 3]           pshb
00AF 36        [ 3]           psha
00B0                          pshd
00B0 37        [ 3]           pshb
00B1 36        [ 3]           psha
00B2                          pshd
00B2 37        [ 3]           pshb
00B3 36        [ 3]           psha
00B4 1830      [ 4]           tsy                         ; initialize y as the new stack frame pointer.
00B6 18EC0A    [ 6]           ldd          Num0,y         ; load the upper 16-bits of the dividend.
00B9 CDA30E    [ 7]           cpd          Denm,y         ; is the divisor>the upper 16-bits of dividend?
00BC 2507      [ 3]           blo          Div32x16a      ; yes. use a fractional divide on initial value.
00BE CDEE0E    [ 6]           ldx          Denm,y         ; load the divisor into x.
00C1 02        [41]           idiv                        ; divide the upper 16 bits by the divisor.
00C2 CDEF00    [ 6]           stx          Quo0,y         ; save the partial quotient.
00C5 CDEE0E    [ 6]Div32x16a  ldx          Denm,y         ; load the divisor into x.
00C8 03        [41]           fdiv                        ; resolve remainder into a 16-bit fractional part.
00C9 CDEF02    [ 6]           stx          Quo2,y         ; save the partial result.
00CC 18ED04    [ 6]           std          Rem,y          ; save the remainder of the fractional divide
                                                            (partial remainder).
00CF 18EC0C    [ 6]           ldd          Num2,y         ; get the lower 16-bits of the dividend.
00D2 CDEE0E    [ 6]           ldx          Denm,y         ; get the denominator again.
00D5 02        [41]           idiv                        ; resolve the remaining quotient.
00D6 18E304    [ 7]           addd         Rem,y          ; add the previous remainder to this remainder.
00D9 18ED04    [ 6]           std          Rem,y          ; save the total remainder.
00DC 8F        [ 3]           xgdx                        ; put last partial quotient into d-accumulator...
                                                          ; & save the total remainder in x.
00DD 18E302    [ 7]           addd         Quo2,y         ; add partial quotient to the lower 16-bits of the
                                                            quotient.
00E0 18ED0C    [ 6]           std          Num2,y         ; save the result.
00E3 18EC00    [ 6]           ldd          Quo0,y         ; get the upper 16-bits of the quotient.
00E6 C900      [ 2]           adcb         #0             ; add the possible carry to the lower 8-bits.
00E8 8900      [ 2]           adca         #0             ; add the possible carry to the upper 8-bits.
00EA 18ED0A    [ 6]           std          Num0,y         ; save the result.
00ED 8F        [ 3]           xgdx                        ; get the total remainder back into d.
00EE CDA30E    [ 7]           cmpd         Denm,y         ; is the total fractional remainder > the divisor?
00F1 2519      [ 3]           blo          Div32x16b      ; no. we’re finished.
00F3 18A30E    [ 7]           subd         Denm,y         ; yes. It will be < than 2 * Divisor.
00F6 18ED04    [ 6]           std          Rem,y          ; save the final remainder.
00F9 18EC0C    [ 6]           ldd          Num2,y         ; now we must add 1 to the 32-bit quotient.
00FC C30001    [ 4}           addd         #1             ; add 1 to the lower 16-bits.
00FF 18ED0C    [ 6]           std          Num2,y         ; save the result.
0102 18EC0A    [ 6]           ldd          Num0,y         ; get the upper 16-bits.
0105 C900      [ 2]           adcb         #0             ; add the possible carry to the lower 8-bits.
```

```
536 0107 8900       [ 2]           adca        #0          ; add the possible carry to the upper 8-bits.
537 0109 18ED0A     [ 6]           std         Num0,y      ; save the result.
538 010C 18EC04     [ 6]Div32x16b  ldd         Rem,y       ; get the final remainder.
539 010F 18ED0E     [ 6]           std         Denm,y      ; overwrite the divisor.
540 0112 30         [ 3]           tsx                     ; need to do this for rtd to work correctly. See
                                                             NOTE.
541 0113                           rtd         x,LocSize   ; deallocate locals & return.
542 0113 C606       [ 2]           ldab        #LocSize    ; number of bytes to deallocate.
543 0115 3A         [ 3]           abx                     ; add it to the current stack frame pointer.
544 0116 35         [ 3]           txs                     ; deallocate storage by updating stack pointer.
545 0117 38         [ 5]           pulx                    ; restore the previous stack frame pointer.
546 0118 39         [ 5]           rts                     ; return to the calling routine.
547                 *
548                      cycles total=347                  ; Total number of E cycles for a 32 x 16 divide.
549                 *
Errors:  None
Labels: 30
Last Program Address: $0118
Last Storage Address: $FFFF
Program Bytes: $0119 281
Storage Bytes:  $000D 13
```


freescale™ semiconductor

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com