MIGRATING TO THE 68HC12 IN C

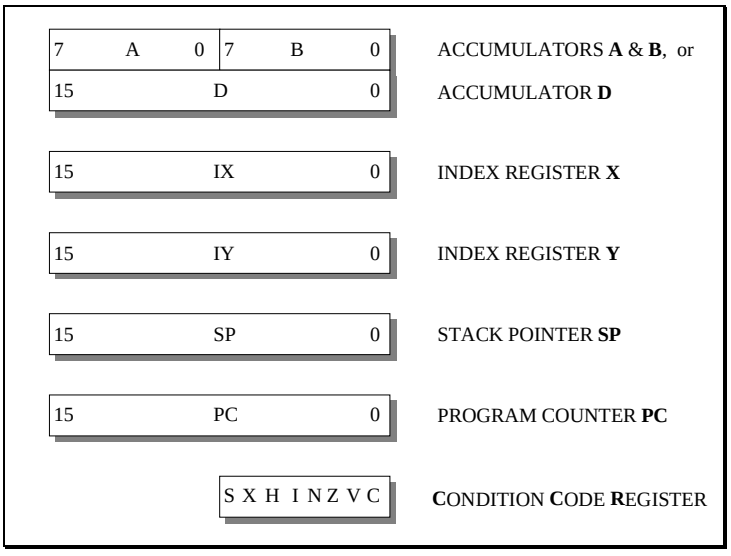
by Jean-Pierre Lavandier (Cosmic Software)
and Greg Viot (Motorola)

INTRODUCTION

An important design goal of the 68HC12 was to maintain software compatibility with the 68HC11 allowing a smooth upward migration from the 68HC11 to the 68HC12. The compatibility goal was achieved by keeping the same programmers model (see figure 1) and the same functional definition of all 68HC11 instructions.

Also, since many microcontroller programmers are moving from assembly to C to improve productivity and maintainability, the 68HC12 designers strengthened the 68HC11 instruction set and addressing modes to better support high level languages. In this paper, we discuss many of the features added to the 68HC12 which allow for efficient C code. We also provide code examples of both microcontrollers for comparison.

FIGURE 1. Identical Programmer's Model for 68HC11 & 68HC12



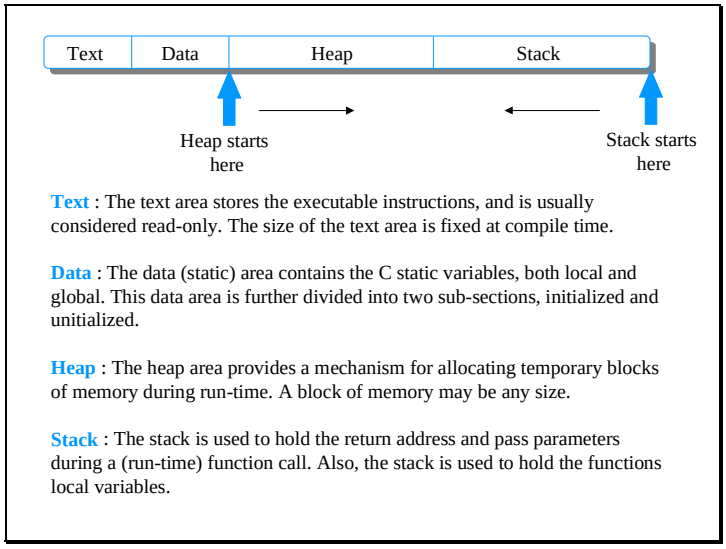
THE C LANGUAGE

The C language basically handles three kinds of objects:

- Static Variables
- Dynamic Variables
- Functions

The terms `static & dynamic' are used here instead of 'global & local'. A variable which is statically allocated has a known address, but is not always global because its scope may be restricted to a function or a file by the C keyword 'static'. Such a variable becomes local, although not dynamically allocated. A dynamically allocated variable is created during the program execution and has an undefined address until it is allocated. Figure 2 shows the memory organization of a C program, including a text (program) area, static data area, and a heap and stack for dynamic variables.

FIGURE 2. Memory Organization of a C Program



A function is a piece of code starting with a label, and ending with a return instruction, such as **rts**. Such a function will be entered by a **jsr** instruction. Operations on a function are restricted to taking its address and calling it with arguments. The address of a function becomes a pointer to that function. A function can be called either directly, using its name, or through a pointer to a function, using the following operations:

```
ldx   #func       ; take address
jsr   func        ; direct call
jsr   0,x         ; indirect call
```

Note that the 68HC12 allows all function calls to become relative by using the PC relative addressing mode:

```
jsr   func,pcr    ; relative call
```

This method allows Position Independent Code (PIC), which can be moved anywhere in memory to be executed. PIC is more useful for RAM-based systems than ROM-based, which is the case for most embedded applications. This is also a useful feature when implementing flash programming routines, which need to be executed from any RAM space. Such routines may be written with the main application, and simply copied in RAM before being executed.

Static variables are allocated once at a known address in the memory space, and generally accessed through a label. On the other hand, dynamic variables are allocated during the program execution, generally at a function entry, and are destroyed (deallocated) when they are no longer needed (at a function exit). Dynamic variables are allocated on the stack, allowing recursivity as required by the C standard. Basic operations on memory variables can be narrowed to:

- take a variable address
- load a variable in a register
- store a register into a variable

Direct operations on memory variables are generally available, avoiding the Load/Operate/Store sequence. For static variables, basic operations can be implemented using the following instructions:

```
ldx   #svar       ; load an address
ldd   svar        ; load a variable
std   svar        ; store a variable
```

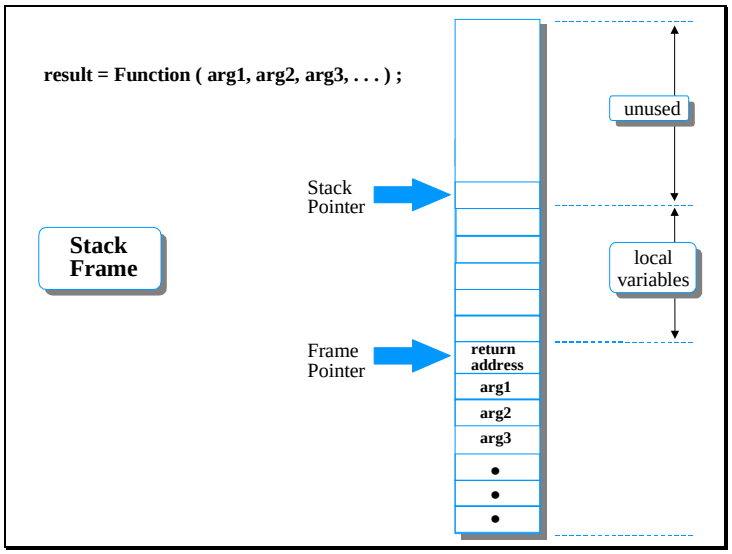
For dynamic variables, we need a way to address an object in the stack. For reference, a stack frame is shown in Figure 3. The 68HC11 cannot address directly an object on the stack, it is then necessary to copy the stack pointer into an index register X or Y, and then access the variable:

```
tsx               ; load stack address
ldd   2,x         ; load and
std   4,x         ; store a variable
```

Computing the address of such a variable still needs more instructions:

```
tsx               ; load stack address
xgdx              ; in the D register
addd  #4          ; add offset
```

FIGURE 3. Sample Stack Frame in C



The need to reload the stack pointer into an index register before any dynamic object access may extend the code size for large functions. Even if redundant loads can be removed by an optimizer, it may be more useful to load the stack address once into a dedicated index register at the function entry. This eases the memory accesses, but complicates an address calculation, as the previous sequence destroys the index register. This problem may be solved by adding in the stack a word containing the stack address, which can be easily computed at the function entry. It costs time and code at function entry, but simplifies the function body and exit.

On the other hand, the 68HC12 makes these accesses easy and efficient as the stack is directly addressable:

```
ldd   2,sp        ; load and
std   4,sp        ; store a variable
```

and a stack address is easily computed by a single instruction:

```
leax  4,sp        ; load effective address
```

The stack frame allocation is needed to create space on the stack the space needed to host the dynamic variables, and to initialize a 'frame pointer' allowing a direct access to this area. For the 68HC11, the stack allocation can be implemented by the following sequence:

```
tsx               ; get stack pointer
xgdx              ; in the D register
subd  #size       ; move backward
xgdx              ; and store back
txs               ; in the stack pointer
```

The same sequence will be used at the function end to deallocate the stack frame, by adding its size to the stack pointer. The 68HC12 'load effective address' will solve the problem, as only one instruction is needed to open and close the stack frame:

```
leas  -size,sp          ; open stack frame
```

Not using a dedicated frame pointer is more practical on the 68HC12 because the stack is directly addressable. Not using a frame pointer saves a word on the stack, as using a dedicated register for a frame pointer forces the compiler to save and restore it at each function entry and exit. This also leaves both index registers available for code generation. With this method, variables on the stack, including arguments, will be accessed with positive offsets from the stack pointer, thus loosing half of the efficient encoding possibilities of the indexed addressing mode, allowing offsets between -16 and +15 to be coded on one post-byte. Entry and exit sequences consist only of 'leas' instructions to open and close the stack frame.

A second choice is to use a frame pointer. It has to be saved on function entry and the current frame pointer has to be computed. It will have to be restored at the function exit. Entry and exit sequences may look like:

```
entry:
      pshx              ; save old FP
      tfr   sp,x        ; set new FP
      leas  -size,sp    ; open stack frame
      ...
exit:
      leas  size,sp     ; clean-up stack frame
      pulx              ; restore old FP
      rts               ; return to caller
```

Functions with small stack frames can take advantage of the 68HC12 auto-increment/decrement addressing modes:

```
entry:
      stx   size+2,-sp  ; open frame and save old FP
      leax  size+2,sp   ; set new FP
      ...
exit:
      ldx   size+2,sp+  ; clean stack and restore FP
      rts               ; return to caller
```

For auto-inc/decrement modes, the maximum value for the offset being 8, this method can be used for stack frames up to 6 bytes, the +2 offset representing the FP size. The frame pointer is set between arguments and dynamic variables, so both positive and negative offsets can be used efficiently to access stack.

The C language basically handles four kinds of variables:

- integers
- reals
- pointers (and arrays)
- structures (and unions)

Integers may be 8-bit, 16-bit or 32-bit. As bit size becomes larger, it becomes more difficult to handle integers efficiently. The registers available in both processors are **D**, **X** and **Y**. Register **D** is a 16-bit register, dividable in **A** and **B**, two 8-bit registers, and supporting arithmetic and logic operations, either directly or through two 8-bit operations.

FIGURE 4. Summary of Addressing Modes for 68HC12

Addressing Mode	Source Format	Abbreviation	Description
Inherent	INST	INH	Operands (if any) are in CPU registers
Immediate	INST #opri8/16	IMM	Operand is included in instruction stream, 8 or 16-bits
Direct	INST opr8a	DIR	Postbyte is lower 8-bits of address $0000-$00FF
Extended	INST opr16a	EXT	Postbytes form 16-bit address
Relative	INST rel8/16	REL	8 or 16-bit relative offset from PC
Indexed 5-bit offset	INST oprx5,xysp	IDX	5-bit signed offset from X, Y, SP or PC
Indexed pre-decrement	INST oprx3,-xys	IDX	Auto pre-decrement X, Y or SP by 1 through 8
Indexed pre-increment	INST oprx3,+xys	IDX	Auto pre-increment X, Y or SP by 1 through 8
Indexed post-decrement	INST oprx3,xys-	IDX	Auto post-decrement X, Y or SP by 1 through 8
Indexed post-increment	INST oprx3,xys+	IDX	Auto post-increment X, Y or SP by 1 through 8
Indexed accumulator offset	INST abd,xysp	IDX	Accumulator offset A, B, or D from X, Y, SP or PC
Indexed 9-bit offset	INST oprx9,xysp	IDX1	9-bit signed offset from X, Y, SP or PC
Indexed 16-bit offset	INST oprx16,xysp	IDX2	16-bit offset from X, Y, SP or PC
Indexed-indirect 16-bit offset	INST [oprx16,xysp]	[IDX2]	16-bit offset from X, Y, SP or PC, with indirection
Indexed-indirect D accum. offset	INST [D,xysp]	[D,IDX]	Accumulator D offset from X, Y, SP or PC with indirection

Registers **X** and **Y** are basically 16-bit pointers, and support a restricted subset of arithmetic operations only. These registers can be used as indexes to access memory. In the 68HC11, **X** and **Y** are not completely equivalent, as using **Y** cost in most of the cases an extra code byte, and an extra cycle. There only one indexed addressing mode using an unsigned byte offset, allowing a direct access to the 256 bytes from the value of the index register.

The only direct arithmetic operations available are increment and decrement. All other arithmetic and logic operations still can be used with the exchange instructions with the **D** register, but needing two extra instructions to perform the operation.

In the 68HC12, **X** an **Y** are completely equivalent, and can be used in several indexed addressing modes (see figure 4 for addressing modes).

The 'lea' (load effective address) instruction allows all addition operations with constants of any value, and with any value loaded in the **D** register. Subtraction from constant is directly possible, while subtraction from the **D** register need it to be negated, costing extra instructions. Other arithmetic and logic operations are still possible using the exchange instruction with the **D** register.

With the 68HC11, the implementation of a 32-bit integer depends on the use or not of a frame pointer. If no frame pointer is used, a 32-bit integer can be implemented in the register pair **D** and **X**, thus leaving the **Y** register for any memory indirection. If a frame pointer is used, it is dangerous to use the other index register with **D** to host the 32-bit value as this leaves no available pointer if a memory indirection is need, for instance to store the 32-bit result in memory through a pointer. In this case, a 32-bit value will be completely hold in memory, or a word in memory and a word in the **D** register. If these extra words are statically allocated, they become part of the processor context, and need to be saved when an interrupt occurs. Allocating these words on the stack will consume more stack space as they extend the stack frame of any function using 32-bit expressions, but there will be no need to save them on an interrupt.
With the 68HC12, there is no need for a frame pointer, so a 32-bit value can be held in the register pair **D** and **X**, thus leaving **Y** available for a memory indirection.

32-bit operations will be implemented either with inline code if the instruction set allows a code efficient production, or with a function call to a library routine performing the operation. These routines will operate on a first operand located in the **D** and **X** register pair, and a second operand pointed by the **Y** register.

Real numbers supported in C are 32-bit (float) or 64-bit large (double). 32-bit reals will be implemented as 32-bit integers, but 64 bit reals will always be handled directly in memory. All arithmetic operations on reals are implemented with library calls. 32-bit values can use the same argument convention as for the 32-bit integers. 64-bit values will be handled by library routines receiving both operand addresses in the two index registers **X** and **Y**.

Pointers will obviously be implemented in index registers **X** and **Y**, as they are the only ones allowing memory indirection. This does not stop the compiler to still use **D** when no memory indirection is required, for a pointer assignment for instance. The 68HC12 addressing modes offer efficient possibilities for a wide range of situations:

Array Indexing:

```
i = tab[j];       ldx   j           ; load index
                  ldd   tab,x       ; load array element
                  std   i           ; store result
```

This construct can be used with the 68HC11 only if the array is located in the first 256 bytes (zero page). Otherwise, the following code is needed:

```
                  ldd   j           ; load index
                  addd  #tab        ; add array address
                  xgdx              ; result in X
                  ldd   0,x         ; load array element
                  std   i           ; store result
```

Pointer Indexing:

```
i = p[j];         ldx   p           ; load pointer
                  ldd   j           ; load index
                  ldd   d,x         ; load array element
                  std   i           ; store result
```

This addressing mode is not available with the 68HC11, and the following code is needed:

```
                  ldd   j           ; load index
                  addd  p           ; add pointer
                  xgdx              ; result in X
                  ldd   0,x         ; load array element
                  std   i           ; store result
```

Auto-Increment:

```
i = *p++;         ldx   p           ; load pointer
                  ldd   2,x+        ; load value and increment X
                  stx   p           ; store back pointer
                  std   i           ; store result
```

This addressing mode is not available with the 68HC11 and here is one possible solution:

```
                  ldx   p           ; load pointer
```

```
                    ldd   0,x         ; load value
                    inx               ; increment
                    inx               ; register
                    stx   p           ; store back pointer
                    std   i           ; store result
```

The 68HC12 offers indexed-indirect addressing modes making easy the usage of a pointer when it is allocated on the stack, for instance:

```
      i = *p;       ldd   [2,sp]      ; load value directly
                    std   i           ; store result
```

The 68HC11 needs the following sequence:

```
                    tsx               ; get stack address
                    ldx   2,x         ; load pointer
                    ldd   0,x         ; load value
                    std   i           ; store result
```

The form allows efficient access to an array of pointers with an index:

```
      i = *ptab[i]; ldx   #ptab       ; load array address
                    ldd   i           ; load index
                    lsld              ; align to word offset
                    ldd   [d,x]       ; load value directly
                    std   i           ; store result
```

The 68HC11 needs the following sequence:

```
                    ldd   i           ; load index
                    lsld              ; align to word offset
                    addd  #ptab       ; add array address
                    xgdx              ; result in X
                    ldx   0,x         ; load pointer
                    ldd   0,x         ; load value
                    std   i           ; store result
```

All the above examples for the 68HC11 use the **X** register. If the **Y** register was used, the actual code would be still larger as instructions handling the **Y** register need one extra byte (and cycle).

Structures can only be copied into another structure variable, or copied onto the stack to be passed as argument This block move may be implemented differently depending on the structure size. The 68HC12 offer efficient 'move' instructions which can be used for that purpose, along with

the loop instructions. Assuming that source and destination addresses are loaded respectively in the index registers **X** and **Y**, a structure copy may be implemented by the following sequence:

```
      ldd   #size       ; structure size
loop:
      movb  1,x+,1,y+   ; copy one byte and increment regs
      dbne  d,loop      ; count down and loop back
```

It is possible to enhance this copy by using a **move word**:

```
      ldd   #size/2     ; structure size as word count
loop:
      movw  2,x+,2,y+   ; copy one word and increment regs
      dbne  d,loop      ; count down and loop back
(     movb  0,x,0,y     ; copy last byte if size is odd )
```

The last **move byte** is needed only if the structure size is odd.

Stacking a structure can be implemented in a similar way, assuming that the source address in loaded in the **X** register (even size):

```
      leax  size,x      ; move to the end
      ldd   #size/2     ; structure size as word count
loop:
      movw  2,-x,2,-sp  ; stack one word and decrement regs
      dbne  d,loop      ; count down and loop back
```

Those sequences can be implemented in such a way only because the 'move' instructions does not need the **D** register. It is not possible to use the same method with the 68HC11, or only if the structure size is smaller than 256 bytes (usually enough):

```
      ldaa  #size       ; structure size
loop:
      ldab  0,x         ; load one byte
      stab  0,y         ; store it
      inx               ; increment source pointer
      iny               ; increment destination pointer
      deca              ; count down
      bne   loop        ; and loop back
```

This sequence cannot be optimized using word moves as all the processor registers are already used.

Stacking the structure with a 68HC11 can be implemented with the following sequence:

```
      ldab  #size       ; structure size
      abx               ; move to the end
loop:
```

```
        dex             ; decrement source pointer
        ldaa  0,x       ; load one byte
        psha            ; push it
        decb            ; count down
        bne   loop      ; and loop back
```

with no possible word optimization.

EXPRESSIONS

In most of the cases, a C expression is turned into a sequence of loads into registers, operations on registers and stores into memory. The compiler often has to enlarge a result, to match the C rules for expression evaluation, mainly from char to int. This operation named 'widening' depends on the char type. An unsigned char will be simply widened by adding a zero as Most Significant Byte (MSB). A signed char will be widened by a sign extend operation, which add either a `$00` or a `$FF` as MSB. An instruction is provided with the 68HC12 to ease this process, assuming a byte has been loaded in the B register:

```
        tfr   b,d   ; clear the A register for an unsigned extension
        sex   b,d   ; sign extend B into A for a signed extension
```

If the first operation can be simply replaced by a simple **clra**, the second operation is more difficult to implement with a 68HC11:

```
        clra            ; clear A first
        tstb            ; test byte
        bpl   pos       ; skip if positive
        coma            ; turns A to $FF otherwise
pos:
```

Note that the ANSI standard leaves the compiler free to choose the default type of a 'char' variable; it is selected as 'unsigned char' by default as there is no efficient sign extend instruction.

In complex expressions, intermediate results need to be stored, either in memory, or in another register, depending on the type of the operation. For instance, such an expression:

```
        a = (b & c) | (d & e);        /* all char's */
```

cannot be translated without saving an intermediate result. The 68HC12 addressing modes allow such an operation to be done efficiently:

```
        ldab  b         ; load first operand
        andb  c         ; operates with second operand
        pshb            ; save first result on the stack
        ldab  d         ; load third operand
        andb  e         ; operates with fourth operand
        orab  1,sp+     ; last operation and stack off
        stab  a         ; store result
```

The post-incremented indexed addressing mode allows the operation to be performed, and the stack to be cleaned at the same time. If the same operation has to be implemented with the 68HC11, the sequence becomes:

```
        ldab  b     ; load first operand
        andb  c     ; operates with second operand
```

```
pshb        ; save first result on the stack
ldab  d     ; load third operand
andb  e     ; operates with fourth operand
tsx         ; load stack address
orab  0,x   ; last operation
ins         ; clean-up stack
stab  a     ; store result
```

Those costless temporary cells can be used for inlining multiplications by specific constants, such as the following sequence multiplying the **D** register by 3:

```
pshd        ; copy value on the stack
lsld        ; left shift = multiply by 2
addd  2,s+  ; add value = multiply by 3, and clean-up stack
```

Except the fact that all the arithmetic and logic instructions accept the new indexed addressing modes, there are no big changes in the basic operations. The 68HC12 provides more efficient register transfer operations than the 68HC11, where transfers between the index registers **X** or **Y** and the **D** register were only possible with exchange instructions. The main enhancements are with the multiply and divide operations.

The 68HC12 provides four multiply instructions:

```
mul         ; unsigned 8 x 8 -> 16, same as 68HC11's
emul        ; unsigned 16 x 16 -> 32
emuls       ; signed 16 x 16 -> 32
emacs       ; signed 16 x 16 -> 32 with addition in memory
```

The **emul** instruction can be used to implement the 16-bit multiplication as usually used in C, assuming all variables are 'int':

```
a = b * c;        ldd   b     ; load first operand
                  ldy   c     ; load second operand
                  emul        ; operate
                  std   a     ; store result
```

The **emul** instruction always provides a 32-bit result in registers **D** and **Y**, but in this example, the lower 16-bit word of the result (in the **D** register) only is stored back. The high word (in register **Y**) is simply discarded. Note that the unsigned multiply instruction is used even if operands are signed because the result is truncated to the lower 16 bits of the product, and because **emul** cost one byte only, instead of two for **emuls**.

Note also that this instruction is very fast as it costs only 3 cycles. It means that the usual optimizations replacing multiplications with special constants (power of two, small values) by faster sequences (left shifts, push/add) may not be necessary. Using the 'emul' instruction may be as fast, or even faster than the equivalent sequence, but it will need the **Y** register, thus assuming that it is not used for something else.

To get a full 32-bit result, the C syntax has to be used carefully, because writing the expression:

```
la = b * c;   /* long = int * int */
```

will get a 16-bit result from the product of b and c, will sign extend it to a long before to store the 32-bit result, as required by the C evaluation rules. The proper syntax is:

```
la = (long) b * c;      ldd   b     ; load first operand
                        ldy   c     ; load second operand
                        emuls       ; operates
                        std   a+2   ; store result LSW
                        sty   a     ; store result MSW
```

The explicit conversion applied to one operand (either b or c) forces the compiler to produce a 32-bit result for the multiplication, allowing it to use efficiently the **emuls** instruction.

The **emul** or **emuls** instructions can be used to provide a 32-bit result only if operands (b and c) are both signed or both unsigned.

The last multiplication instruction allows a 32-bit result to be accumulated in memory. To allow the compiler to use directly this instruction, the C syntax must be:

```
la += (long) b * c;     ldx   #b    ; load first operand address
                        ldy   #c    ; load second operand address
                        emacs la    ; operates
```

The 68HC12 provides five divide instructions:

```
idiv        ; unsigned 16 : 16, same as 68HC11
fdiv        ; fractional unsigned 16 : 16, same as 68HC11
idivs       ; signed 16 : 16
ediv        ; unsigned 32 : 16
edivs       ; signed 32 : 16
```

The two first instructions are directly accessed by the C expressions:

```
a = b / c;              ldd   b     ; load first operand
                        ldx   c     ; load second operand
                        idivs       ; operates
                        stx   a     ; store result (quotient)
```

The **idiv** instruction is used when operands are unsigned. The two last divisions can be used only with the appropriate C syntax:

```
a = (int)(la / b);      ldd   la+2  ; load first operand LSW
                        ldy   la    ; load first operand MSW
                        ldx   b     ; load second operand
                        edivs       ; operates
                        sty   a     ; store result (quotient)
```

The explicit conversion is necessary to force the compiler to get a 16-bit result from the division, thus allowing it to use the **edivs** instruction. Otherwise, the division provides a 32-bit result, as one of its operands is a 32-bit value, and the result is truncated by the assignment. As for the multiplications, the **ediv** and **edivs** can be used only if operands are both signed or both unsigned.

The 68HC12 also provides specific instructions for computing the min or max of two unsigned values, on 8 or 16 bits. There is no special C operators for such operations, so the best solution is

to support these instructions with predefined C functions allowing the compiler to use directly these instructions. These instructions work only with indexed addressing mode, so they are directly operational for dynamic variables, so assuming that variables a, b, and c are allocated on the stack:

```
a = max(b, c);     ldd   2,sp  ; load first operand
                   emaxd 4,sp  ; maximize with second operand
                   std   6,sp  ; store result
```

or when used for accumulation in memory:

```
a = max(a, b);     ldd   2,sp  ; load first operand
                   emaxm 4,sp  ; maximize second operand in memory
```

STATEMENTS

A C program flow is driven by if-else statements, loop statements (while, do, for), and switch statements.

The 68HC12 provides relative conditional branches with 16-bit offsets, allowing the whole memory to be reached. It provides also a set of special branches:

```
tbeq        test and branch if equal to zero
tbne        test and branch if not equal to zero
ibeq        increment and branch if equal to zero
ibne        increment and branch if not equal to zero
dbeq        decrement and branch if equal to zero
dbne        decrement and branch if not equal to zero
```

These instructions operate on a register which may be **A**, **B**, **D**, **X**, **Y** or **SP**. They can be used to implement loops, but also compare and branch sequences for specific compare values:

```
if (a == 1)             ldd   a           ; load value
      ...               dbne  d,endif     ; decrement and
                              ...         ; branch if zero
                        endif:
```

The value returned by a function can be tested against zero with the following sequence:

```
if (func() != 0)        jsr   func        ; result in D
                        tbeq  d,endif     ; branch if D is zero
```

Using these instructions directly in C loops is interesting when the counter variable is in a register. If it is a memory variable, its usage is less easy as the result need to be written back in memory. This can be achieved by duplicating the store instruction on top of the loop body, and after the branch.

These instructions will be very useful in built-in loops, such as those needed for structure copies, or shift operations:

```
a = b << c;         ldd   b           ; load value
                    ldx   c           ; load count
                    beq   nosh        ; skip if count is zero
              loop:
```

```
                lsld              ; shift value
                dbne  x,loop      ; count down and loop back
          nosh:
```

The switch statement is also a C instruction which can be efficiently implemented with the 68HC12 instructions. Small lists of consecutive values can be implemented using the decrement and branch instructions:

```
switch (i)        ldd   i           ; load value
      {           tbeq  d,case_0    ; branch here if D is zero
case 0:           dbne  d,case_1    ; branch here if D was on
      ...         dbne  d,case_2    ; branch here if D was two
case 1:           ...
      ...
case 2:
      ...
```

Each entry costs 3 bytes, but it turns to be not time efficient as the list becomes longer. For longer contiguous lists, another solution is to use the PC relative indexed indirect addressing mode:

```
switch (i)        ldd   i                 ; load value
      {           cpd   #case_last        ; compare with last case
case 0:           cpd   #case_last        ; compare with last case
      ...         bgt   case_default      ; skip to default
case 1:           lsld                    ; align to word offset
      ...         jmp   [d,pc]            ; jump to the right code
case 2:
      ...         dc.w  case_0            ; table of addresses
case 3:           dc.w  case_1            ; for all the
      ...         dc.w  case_2            ; case labels
                  dc.w  case_3
                  ...
```

Each entry costs 2 bytes, with a fixed initial cost for the range check, the shift and the jump instructions. The execution time is also independent from the amount of cases. Non-contiguous cases are easy to handle with the first method, by subtracting the difference with the previous case, and with the second method, by adding dummy entries to fill the holes from the missing cases, which may enlarge the address table size beyond a reasonable limit. If the first case is not zero, the result is biased and checked before to enter the dispatch sequence.

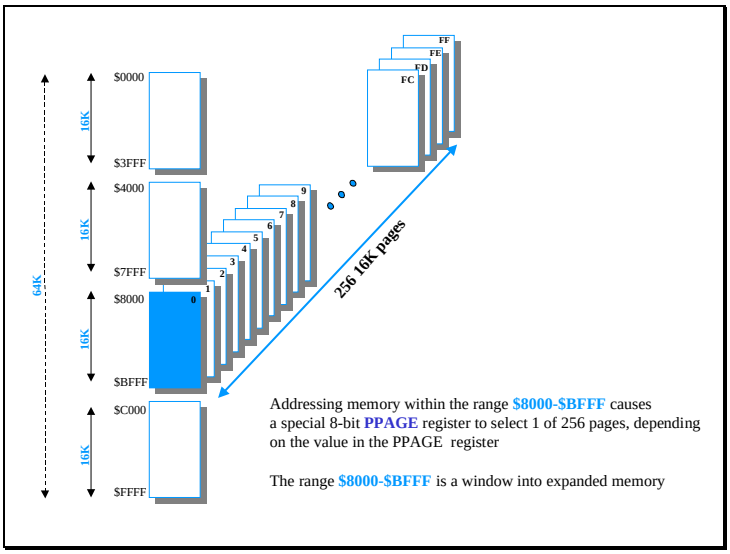
FUNCTIONS

The 68HC12 allows two kinds of function calls. The first one is the same as the 68HC11, using the pair **jsr/rts**, working with 2-byte addresses, allowing a direct access to 64K of code. The second kind uses the pair **call/rtc**, working with 3-byte addresses, allowing a direct access to the full extended code memory, also called bank switching. A 16K window in the 64K addressing space is mapped to the expected bank by setting the specific register **PPAGE** in the I/O space. Figure 5 illustrates banks extending addressing beyond 64K.

Calling a bank switched function needs the processor to update the **PPAGE** register with a new value, while saving the previous value for the return. These two operations being merged into one single instruction, the compiler is able to call directly (or thru a pointer) a banked switched function. Otherwise, the compiler has to use a small piece of code located in a non-switched area of the application to perform all saving and switching operations. The **call** instruction pushes 3 bytes, 2 bytes for the 16-bit return address, and one byte for the current content of the **PPAGE** register, and jumps to the target location after loading the **PPAGE** register with the new value. The **rtc** instruction will restore both the **PPAGE** and the **PC** registers from the values on the stack.

Such a mechanism is simple and efficient, but implies a few constraints. A bank has a limited size, and a switched function cannot cross a bank boundary, thus limiting the size of a bank switched function to 16K. This is large enough for one function, but when packing several functions in one bank, there will be a hole at the end. A good linker will help the functions allocation by ordering the functions in order to have the smallest holes as possible.

FIGURE 5. Extending Beyond 64K through Bank Switching



If variables are allocated in ROM with the code (const variables for example) they will be accessed thru the same decoding mechanism as the code itself, so using the value of the **PPAGE** register. This means that such a variable cannot be accessed from another bank, where the **PPAGE** register have a different value. We can consider these variables as 'static' variables whose scope is limited to a bank.

A switched function has to be called using a **call** instruction. It then has to be exited using a **rtc** instruction. If such a function has to be called from the same bank, its exit sequence forces the compiler to still call it with a **call** instruction, even if a **jsr** would have been enough. This means that a function has only one way to be called. This does not stop a function which is called only from the same bank to be called directly by a **jsr** instruction if such an efficiency is actually needed. It may become difficult to declare properly all these functions if the application size

grows, and it has been growing if the bank switching mechanism was needed. The extra cost of defaulting all functions to be banked may be small enough to allow the application to still be kept efficient.

CONCLUSION

By extending the 68HC11 addressing modes and instruction set, the 68HC12 processor allows a C compiler to produce a tighter code. Combined with a fast instruction execution, C code is smaller and much faster. The bank switching mechanism allows applications to grow beyond the 64K limits with a small extra cost.