Introducing the MC68HC12 (part 1 of 3)

James M. Sibigtroth
Mail drop OE313
Motorola AMCU
6501 William Cannon Dr W
Austin, TX 78735-8598
RFTP70@email.sps.mot.com

Like any truly general purpose microcontroller, the new MC68HC12 family is going to mean a lot of different things to a lot of different users. To developers of battery operated applications, the 68HC12 will mean low voltage (2.7 volts) and low power consumption. To 68HC11 users it will mean a much higher performance MCU with a rich superset of HC11 instructions. To developers of larger programs written in C, the 68HC12 will mean very efficient high-level language programs and easy access to more than 4 megabytes of program space. And everyone will like the new single-wire background debug mode. This article describes the overall features of the first two derivatives of this family and compares the CPU12 to the M68HC11 CPU. Subsequent articles in this series will provide greater detial on some of the most interesting new features of this new MCU family.

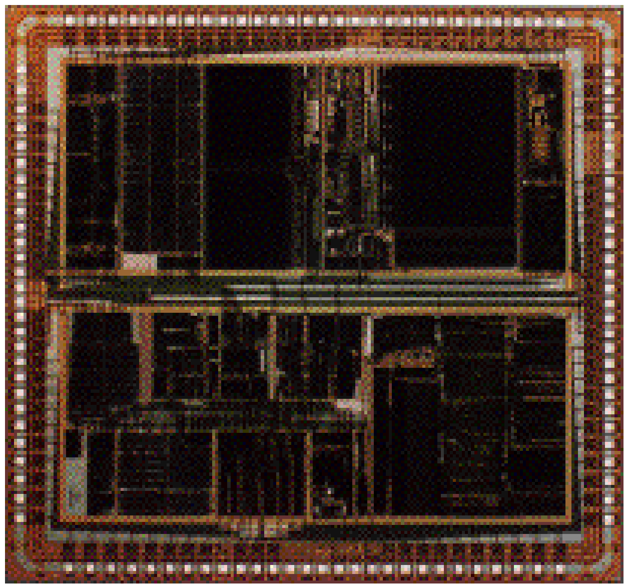
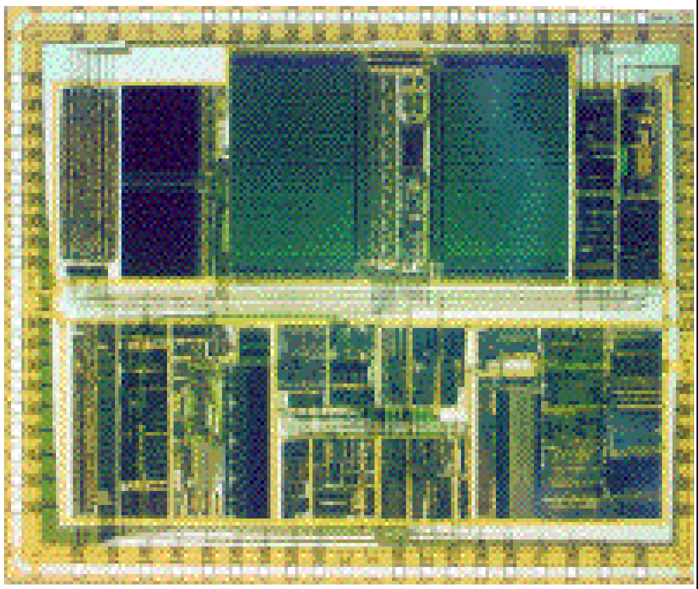
The Seeds of a New Generation of MCUs

The first M68HC12 device to reach silicon was the 112-pin MC68HC812A4 which is primarily intended for larger expanded mode systems. The second device, the 80-pin MC68HC912B32, is intended primarily for single-chip applications. Both devices are based on a modular layout and the new 16-bit CPU12 processor which is completely compatible with M68HC11 programs but it is greatly enhanced. The silicon process is designed to operate at 8MHz bus speed over the whole supply range from 2.7 to 5.5 volts and low power design techniques were emphasized throuout the chip.

The most significant difference between the first two derivatives is the expansion bus structure and the on-chip memory. The MC68HC812A4 has 1k bytes of static RAM and 4k bytes of byte-erasable EEPROM but no large program memory. Systems based on this MCU use a non-multiplexed bus to access external program memory. Up to 22 address lines plus 6 programmable chip selects allow access to more than 4 megabytes of program memory and more than 1 megabyte of data memory. Although the MC68HC12 has full 16-bit data paths throughout, the external bus can operate in either a 16-bit wide mode for best performance or an 8-bit narrow mode so single 8-bit wide EPROMs and RAMs can be interfaced for lower cost systems.

The MC68HC912B32 has a large 32k byte flash EEPROM for program memory, 1k bytes of static RAM, and 768 bytes of byte-erasable EEPROM. This is the first MCU to include both flash EEPROM and byte-erasable EEPROM on the same chip. An external 12 volt supply is used to erase and program the flash memory, the byte-erasable EEPROM uses only the normal 2.7 to 5.5 volt supply for programming and erase operations. The MC68HC912B32 can be used in expanded mode systems but its multiplexed 16-bit address/data bus is primarily intended for factory testing.

Features of the first two MC68HC12 derivatives

	
MC68HC812A4	**MC68HC912B32**
16-bit CPU	16-bit CPU
2.7 - 5.5 Volt Operation	2.7 - 5.5 Volt Operation
Programmable PLL or Xtal = 2x Bus Rate	Xtal = 2x Bus Rate
112 pins (up to 95 General Purpose I/O) Key Wakeup on Three 8-Bit Ports	80 pins (up to 64 General Purpose I/O)
Expanded Non-mux Bus or Single Chip 16-Bit Wide or 8-Bit Narrow	Single Chip or Expanded Multiplexed Bus 16/16 Wide or 16/8 Narrow
Memory Expansion to >5 megabytes	
6 Programmable Chip Selects	
4K EEPROM 1K SRAM 256 Byte Register Space	32K Flash EEPROM 768 EEPROM 1K SRAM 256 Byte Register Space
8 Channel, 8-Bit A/D	8 Channel, 8-Bit A/D
8 Channel Timer	8 Channel Timer
16-Bit Pulse Accumulator	16-Bit Pulse Accumulator
	4 Channel PWM
2 Channels Asynchronous SCI	1 Channel Asynchronous SCI
1 Channel Synchronous SPI	1 Channel Synchronous SPI
	1 Channel J1850 Digital Serial
Single-Wire BDM	Single-Wire BDM with Hardware Breakpoints
COP Watchdog Timer and Clock Monitor	COP Watchdog Timer and Clock Monitor
Periodic Interrupt Timer	Periodic Interrupt Timer

Rather than dividing the high speed input clock by four or more, these chips divide by two to avoid higher frequency circuits that draw more power. Using the same 16 MHz crystal, the HC12 gets an 8 MHz bus rate compared to 4 MHz on an HC11. Both the 'A4 and the 'B32 can use WAIT and STOP modes to reduce power and the 'A4 provides additional ways to save power. A PLL can be used with a lower frequency crystal (say 32 KHz) and multiply this up to the desired operating frequency. At times when processing demands are lower, the PLL can be used to reduce the clock speed and thus reduce power consumption. Additional separate clock dividers for the system clock and the peripheral modules allow a program to speed up and slow down the CPU and bus rate while maintaining a constant clock speed for timers and serial intrface peripherals. This allows for program controlled trade-offs between performance and power savings.

Both new parts allow most pins to be used for general purpose I/O pins when they are not needed for other functions. Even bus control pins such as the E clock and R/W can be used for general purpose I/O if there are no external memory devices. This makes a maximum of 95 general purpose I/O pins on the MC68HC812A4 and a maximum of 64 general purpose I/O pins on the MC68HC912B32. A key wake up system on the 'A4 allows selected edges on any combination of pins in three 8-bit ports to trigger system interrupts.

Both parts offer an 8 channel timer similar to the general purpose timer of a 68HC11 except all 8 channels can be configured for input capture or output compare operations and there are new clocking possibilities. The pulse accumulator clock input can be routed to the main 16-bit timer, and since the pulse accumulator can get its clock from an external pin, this same external pin can be used to clock the main timer. Cascading the pulse accumulator counter and the main timer counter can also allow for very long timer count periods. Another clock configuration allows one of the output compare functions to reset the 16-bit main timer counter so you can get a modulo count. Both chips also have a 16-bit pulse accumulator to count events on an external pin or perform gated time accumulation (measure the cummulative time the input pin was in a selected state).

The MC68HC912B32 has 4 PWM channels with separate 8-bit registers to control each pulse width and each duty cycle. A flexible clock select block allows each channel to operate at a different frequency than the others. Rates are derived from taps off of an 8-bit divider followed by 8-bit modulo dividers giving a possible input frequency range of 8 MHz down to about 125 Hz. the PWM outputs can be left-aligned or center-aligned. The center-aligned option results in fewer simultaneous edges and therefore less generated noise.

The 'A4 derivative has two standard asynchronous SCI channels plus a synchronous SPI channel. The 'B32 has one asynchronous SCI channel, one synchronous SPI channel, and a digital J1850 serial data link controller.

The 8 channel 8-bit analog to digital converter (A/D) is similar to the A/D on the M68HC16 family, and has more features than the A/D converters on the M68HC11 family.

The COP watchdog timer, the system clock monitor, and the periodic interrupt are found on all MC68HC12s. The watchdog timer (when enabled) triggers a reset if the application software fails to complete a service sequence within the selected timeout period. The clock monitor (when enabled) triggers a system reset if the clock frequency falls below a trip frequency. The periodic interrupt (sometimes called a real time interrupt) can generate interrupts at a user selected rate. The periodic interrupt rate is divided down from the module clock (M) rate. In the 'B32 the M clock is the same as the E clock rate but in the 'A4, the module clock frequency can be E, E/2, E/4, or E/8. The periodic interrupt rate is selectable as M divided by 2^{13} through M divided by 2^{19} or about 1 ms to 65 ms assuming an 8 MHz bus rate and no extra E to M dividers.

Brain Surgery

Now that we have met the first two members of this new MCU family, we can take a closer look at the brains (the CPU12). First we will look at the similarities between the CPU12 and its predecessor, the M68HC11, and then we will explore its many new features and instructions.

To begin with, the programmer's model and interrupt stacking order for the CPU12 are identical to that of the M68HC11. In addition all source code for the M68HC11 is accepted by CPU12 assemblers with no modifications. Most M68HC11 instructions even assemble to the same object code on the CPU12. All of this means that programmers that are familiar with the M68HC11 will be quite at home programming the CPU12. But these same 68HC11 programmers will quickly discover that the CPU12 does a lot of things they always wished the HC11 could do... and a lot of things they never even thought of.

The most obvious improvement is that the CPU12 is a 16-bit processor with an ALU that is as wide as 20 bits for some operations. All data busses in the M68HC12 are 16-bits and the external bus interface is normally 16 bits although a narrow 8-bit option can be selected to allow a less expensive system with single 8-bit wide external memory devices. Even when this narrow data bus option is chosen, the CPU12 still thinks everything is 16 bits. An intelligent bus interface temporarily stops clocks to the CPU when it needs to split a 16-bit access into two 8-bit pieces.

Next, there is an instruction queue (similar to a pipeline) that caches program information so that at least two more bytes of object code, in addition to the 8-bit opcode, are visible to the CPU at the start of execution for all instructions. This means that all of the information needed to complete most instructions is already in the CPU at the start of the instruction. Because of this, many instructions can execute in a single cycle with no delays for fetching additional program information. Program information is fetched into the instruction queue 16 bits at a time but instructions can be any length from one byte to six bytes. This allows CPU12 object code to be more efficient than typical 16-bit MCUs which require instructions to be multiples of 16 bits in length (so even a NOP instruction would need to be two bytes instead of one). Thanks to logic in the CPU12s instruction queue and microcode, there is no execution time penalty for aligned vs. non-aligned instructions.

The indexed addressing mode on the CPU12 has been greatly enhanced. In the M68HC11, Y-relative indexed instructions typically had an extra prebyte before the opcode so these instructions took one extra byte and one extra cycle compared to the same instruction using X as the reference. In the CPU12, all indexed instructions have an opcode, a postbyte, and 0, 1, or 2 extension bytes to specify index offsets. The postbyte code specifies which index register to use as the base reference and the type of indexed addressing. The M68HC11 had only one type of indexed addressing (unsigned 8-bit offset), and only allowed X or Y to be the reference index register. The CPU12 allows X, Y, SP, or PC to be used as the base index register and has seven types of indexed addressing. The M68HC11s unsigned 8-bit offset is replaced by a signed 5-bit offset mode, a signed 9-bit offset mode, and a 16-bit offset mode. For offsets of -16 through +15, the 5-bit offset is included in the postbyte code so an instruction like LDAA 4,X would be encoded into two bytes of object code (in fact the same object code as the M68HC11). LDAA 4,Y also fits in two bytes of object code rather than the M68HC11s three bytes. For offsets between -256 through +255, the CPU12 has the opcode, the postbyte (which includes the sign bit for the offset), and one extension byte to hold the other 8 bits of the offset. There is also a 16-bit offset mode which uses two extension bytes to hold a 16-bit offset.

The accumulator offset indexed mode allows 8-bit accumulators A or B or the 16-bit D accumulator to be used as an additional offset that is added to the base index register to form the effective address of the instruction's operand. Neither the index register nor the accumulator value is changed. These accumulator offset variations are available for any of the four index reference registers, X, Y, SP, or PC.

The CPU12 has a new form of auto- pre/post increment/decrement by -8 through +8. Previous processors that could do auto-increment/decrement addressing only allowed some of these choices and the amount of increment or decrement was implied from the size of the operand(s) of the instruction. For example, LDA ,X++ would adjust the index by one because the operand was a byte while LDX ,Y++ adjusted the index by two because the operand was a word. In the CPU12, you can say LDAA 5,SP- which loads a byte into accumulator A and then post-decrements SP by 5. This flexability allows the CPU12 to work very efficiently with small data structures. Execution time for these instructions is the same as it would be for the simple no-offset case (LDAA 5,+SP takes only three bus cycles). This sub-mode of indexed addressing allows X, Y, or SP to be used as the base index register, but not PC because that would interfere with the normal sequence of execution of instructions. These instructions are especially useful with move instructions such as MOVW 2,SP+,2,X+ which pulls a word from the stack and stores it at 0,X and automatically post-increments SP and X by 2 each.

Finally the CPU12 added two types of indexed-indirect indexing, D accumulator offset, and 16-bit offset. These instructions use X, Y, SP, or PC as the base index register, form an intermediate address by adding D or a 16-bit offset, fetch the 16-bit value from that address, and finally use this fetched value as the effective address to access the operand of the original instruction. This type of indexing can be used to program computed-GOTO type constructs, to access operands in a position-independent way, or to program some types of case or switch statements in C.

The enhanced indexed addressing (shown in the next table) is one of the strongest features of the CPU12 for all kinds of programmers, but the SP-relative indexing is especially important for C-compilers. The CPU12 also added LEAX, LEAY, and LEAS instructions which provide an easy way to do pointer arithmetic. For example a 5-, 9-, or 16-bit signed constant can be added to (or subtracted from) the index, A, B, or D can be added to the index (D may be thought-of as a signed or unsigned value since it is the same width as the address bus), or the index can be replaced by a value from memory (using indexed indirect modes).

Summary of Indexed Addressing Modes

Indexed Addressing Mode	**Source Code Example**	**Object Code**	**Cycles**
5-bit signed constant offset -16...+15 from X, Y, SP, or PC	LDAA 15,X	A6 0F	3
9-bit signed constant offset -256...+255 from X, Y, SP, or PC	LDAA 255,Y	A6 E8 FF	3
16-bit signed constant offset -32768...+65535 from X, Y, SP, or PC	LDAA 4096,PC	A6 FA 10 00	4
8-bit accumulator (A or B) offset from X, Y, SP, or PC	LDAA B,SP	A6 F5	3
16-bit accumulator offset from X, Y, SP, or PC	LDAA D,X	A6 E6	3
Auto Pre-Increment by +1…+8 using X, Y, or SP	LDAA 8,+SP	A6 A7	3
Auto Pre-Decrement by -1…-8 using X, Y, or SP	LDAA 8,-Y	A6 68	3
Auto Post-Increment by +1…+8 using X, Y, or SP	LDAA 1,X+	A6 30	3
Auto Post-Decrement by -1…-8 using X, Y, or SP	LDAA 4,X-	A6 3C	3
D accumulator indexed-indirect from X, Y, SP, or PC	LDX [D,PC]	EE FF	6
16-bit offset indexed-indirect from X, Y, SP, or PC	LDAA [4096,X]	A6 E3 10 00	6

While CPU12 assemblers accept all old M68HC11 instructions, a few are transparently replaced by an equivalent CPU12 instruction (by the assembler - the user does not have to change anything in the source code). For example, the 'HC11 instruction ABX is replaced by (assembled to) LEAX B,X which performs the identical function even down to the way condition code bits are affected. The 'HC11 had several register-to-register transfers and two register-to-register exchanges. The CPU12 replaced these with a completely general transfer/exchange instruction that uses a postbyte to choose transfer or exchange and to specify which registers are involved. Some of these combinations that involve transfer or exchange of an 8-bit register to a 16-bit register, perform sign extension or zero extension as an added bonus. The transfer A to B and transfer B to A instructions in the 'HC11 affected the Z condition code while the CPU12s general transfer instruction does not, so the CPU12 has additional instructions to do TAB and TBA the same way the M68HC11 did. The M68HC11 had instructions to set or clear specific bits in the condition codes register. The CPU12 replaced these with ANDCC and ORCC instructions that use a mask to specify any combination of bits to be set or cleared in the CCR.

To make the instruction set more orthoganal, extended addressing mode versions of the bit manipulation instructions (BSET, BCLR, BRSET, and BRCLR) were added. Also register stacking instructions now include instructions for pushing and pulling CCR and D registers.

Since this is a full 16-bit CPU, several higher resolution math instructions were added, and the ones that were present on the M68HC11 were sped up considerably. The CPU12 can do 8x8 or 16x16 multiply operations in three bus cycles. Divide operations take 11 or 12 cycles depending upon which divide instruction it is. There are also some specialty math instructions including a 16x16 to 32-bit signed multiply-and-accumulate instruction and table lookup-and-interpolate instructions for tables of 8- or 16-bit entries. Refer to the following table for more details.

Math Instruction Speeds

Mnemonic	Operation	Cycles	Time (@ 8.4 MHz Bus Rate)
MUL	unsigned 8x8 = 16 multiply	3	357 ns
EMUL	unsigned 16x16 = 32 multiply	3	357 ns
EMULS	signed 16x16 = 32 multiply	3	357 ns
IDIV	unsigned 16/16 = 16 divide	12	1.43 µs
IDIVS	signed 16/16 = 16 divide	12	1.43 µs
FDIV	unsigned 16/16 = 16 fractional divide	12	1.43 µs
EDIV	unsigned 32/16 = 16 divide	11	1.31 µs
EDIVS	signed 32/16 = 16 divide	12	1.43 µs
EMACS	signed 16x16 -> 32 multiply and accumulate	13	1.55 µs
TBL	table lookup and interpolate 8-bit table entries	8	952 ns
ETBL	table lookup and interpolate 16-bit table entries	10	1.19 µs

Possibly the most unusual instructions in the CPU12 are its four fuzzy logic instructions which do membership function calculations, rule evaluation with weighted or unweighted rules, and a fourth fuzzy logic instruction that calculates the sum-of-products and sum-of-weights needed to do weighted average defuzzification. These fuzzy instructions allow a complete fuzzy inference kernel to be programmed in about 50 bytes and execute in about 60 microseconds compared to 250+ bytes and 750 microseconds for the same program in the M68HC11. This is better than a 5:1 improvement in program size and better than 10x improvement in speed when compared against a 4 MHz bus rate MC68HC11.

CPU12 improvements continue with 8- and 16-bit memory-to-memory moves that work with all practical combinations of immediate, extended, and indexed addressing modes. These instructions are especially useful in a register-based architecture like the 'HC11 or CPU12 because they allow data movement without using up any CPU registers. Team these moves up with the flexible new auto-increment indexing and a block move routine becomes a trivial programming exercise.

The CPU12 also has a complete set of long branches for signed and unsigned conditional branching. And there is a new group of loop primitive instructions (DBEQ, DBNE, IBEQ, IBNE, TBEQ, and TBNE) which use A, B, D, X, Y, or SP as the loop counter. The loop counter is decremented, incremented, or tested and then a branch is taken on the condition that the counter has reached zero or has not reached zero.

Another group of instructions is included for doing MIN or MAX operations between an accumulator (A or D) and a byte or word sized memory location. There are versions of each that place (overwrite) the result into either the accumulator or the memory location. This results in a total of eight instructions in this group (MINA, MINM, MAXA, MAXM, EMIND, EMINM, EMAXD, and EMAXM).

The CALL and RTC instructions in the CPU12 work with logic in the MCU to allow access to more than 4 megabytes of program memory. This is a compromise between linear addressing and bank switching. Linear addressing is easiest for programming but it makes most instructions longer and slower (to allow the larger address to be specified in instructions). Linear addressing also places a cost penalty on users that do not need more than 64k bytes of system memory. Bank switching schemes are normally difficult to manage because of three main problems. First, the interrupt vectors must either reside in an unpaged memory space or they must be repeated in all banks. Second, interrrupts must be blocked during page switching to avoid unpredictable results in the event an interrupt occured part way through the switching operation (this also adds program space and execution time to perform the extra interrupt blocking and un-blocking). And third, the code that performed the bank switch had to reside in an unpaged portion of memory to avoid modification of the executing program (again this could add overhead to jump to the switching routine).

The CPU12 has a bank switching system but the CALL and RTC instructions eliminate the problems usually associated with such schemes. To fix the first problem, MCUs based on the CPU12 fix the paged block to be the 16k byte area from $8000 through $BFFF. Interrupt vectors are not located in this area and on-chip resources such as RAM, registers, and EEPROM can also avoid this area. An 8-bit register in the on-chip register space holds the bank select number (which specifies address output lines A14 through A22 for accesses to extended program memory). The CPU12 uses dedicated control signals to access this PPAGE register so the CPU does not need to know what the actual address of this register is. The CALL instruction works like a JSR except that the old PPAGE register value is stacked along with the return address and a new instruction-supplied value is written to PPAGE. Since all this happens within an instruction, there is no longer any need to block interrupts and the CALL and the destination can reside anywhere including within different pages of extended memory. The RTC instruction works like JSR except that the old PPAGE value is also restored from a value on the stack. For more info see the third article in this series.

Looking Ahead

The next article in this series will explore the new single-wire background debug system on the M68HC12 family. This system set out to provide more powerful debug features while making the system less intrusive than any previous background system. Getting the ambilical down to a minimum of two conductors (ground and a single communication wire) is a big step, but as you will see in the next article, that was just the beginning.

M68HC12 Development Support (part 2 of 3)

James M. Sibigtroth
Mail drop OE313
Motorola AMCU
6501 William Cannon Dr W
Austin, TX 78735-8598
RFTP70@email.sps.mot.com

In the previous article we got a look at the first two derivatives in Motorola's M68HC12 family of 16-bit microcontrollers. This time we look more carefully at the new single-wire background debug system. This system offers advanced debug features without interfering with the resources of the target application. We also look at some practical uses of this system that can help in the manufacture and maintenance of application systems based on these new MCUs.

Teeny Tiny Interface - Comprehensive Control

In development systems, smaller is better when it comes to the physical and resource requirements to gain access to a target application system. In many products it is unreasonable to connect a large umbilical connector that clamps over the main processor. The desire to debug systems in their final packaged form has led to various schemes that rely on serial communication to minimize the number of connections to the target system. Some of these systems also rely on some sort of background task that runs within the target system to respond to requests and commands that were passed into the system through a serial port. Typically the actual operation of debug commands and the real time operation of the target application are mutually exclusive.

The M68HC12 takes this technology to a new level in both the reduction of the physical interface and greater separation of debug and target application functions. The physical interface for the background system in the 'HC12 is a single MCU pin that does not share functions with any target application functions. Background access can be gained in any 'HC12 target application system by connecting a common ground and this single communication wire. With this trivial connection, the host can read or write any location in the 64 kbyte map of the target MCU without stopping or slowing down the real time operations of the target application. For other debug functions such as reading and writing CPU registers, tracing single instructions in the application code, reading and writing whole blocks of memory, and accessing other development features, the target application can be stopped (forced to an active background debug mode) to wait for serial commands.

Products based on this MCU can be fully assembled before the on-chip flash memory is programmed with target application code. The background debug interface can be used to program or reprogram flash or byte-erasable EEPROM after final assembly. This interface can also be used for maintenance modifications to application code or for product troubleshooting in the field.

Figure 1 shows a typical setup with a host development system connected to a target application. A small pod between the host and the target system accepts commands from the host side through an ordinary RS232 link. The pod contains a small "probe" processor that converts instructions from the host into commands for the custom single-wire serial interface to the BKGD pin of the target MCU. One implementation of this probe has a 6-pin connection from the pod to the target system. Two of the pins are not connected and the other four pins are Vss, target Vdd, BKGD, and RESET. Vdd is optional and can be used to supply a small amount of power to run the probe processor in the pod. RESET is also optional but it allows a convenient way for the host to remotely force a reset in the target system. In a working target system, the host could indirectly force a reset using serial commands, but the hard connection to RESET can force recovery in a target system that was erroneously stuck in stop mode or where runaway code changed the operating frequency of the target. This 6-pin connector is a standard for several Motorola evaluation products, but if your application has special physical requirements it would be a trivial exercise to build a small adapter cable to connect the pod to your application system.

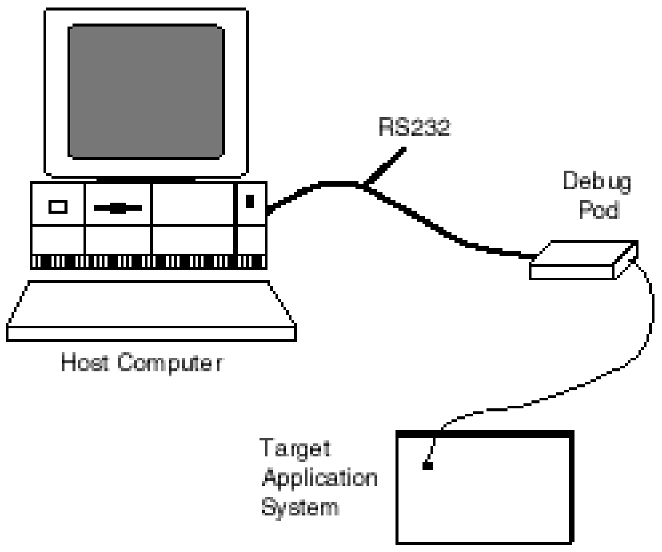


Figure 1 - Typical Background System Hookup

Minimizing the Physical Connection

The primary function of the single interface pin is for high-speed bidirectional data transfer. The physical interface is a pseudo-open-drain connection to avoid conflicts between the host and the target since either can be a driver at different times. As shown in figure 2, timing is based on the target system bus clock. Data frames consist of an 8-bit command and optional additional information depending upon the command. Each bit time is 16 target bus clocks and ones and zeros are differentiated by their low-time. The host initiates each bit time by driving the BKGD pin low, but either the host or the target can be the driver of the bit length depending upon the direction of data. When the host is reading data from the target, the host briefly drives the BKGD pin low (about 2 to 4 target bus cycles) and releases it. The target either allows the pin to return to logic one to indicate a bit value of one, or the target continues to drive the pin low for about 13 cycles (starting from where it first noticed the falling edge from the host) to indicate a value of zero. The host reads the sense of the bit at about cycle 10. Normally an open-drain connection would be relatively slow to account for the R-C time constant of the rising edges but this interface uses brief driven-high pulses to get faster data rates. The resulting interface allows a much wider band of tolerance for variations in host versus target speeds than say the asynchronous protocol used by RS232 terminals.

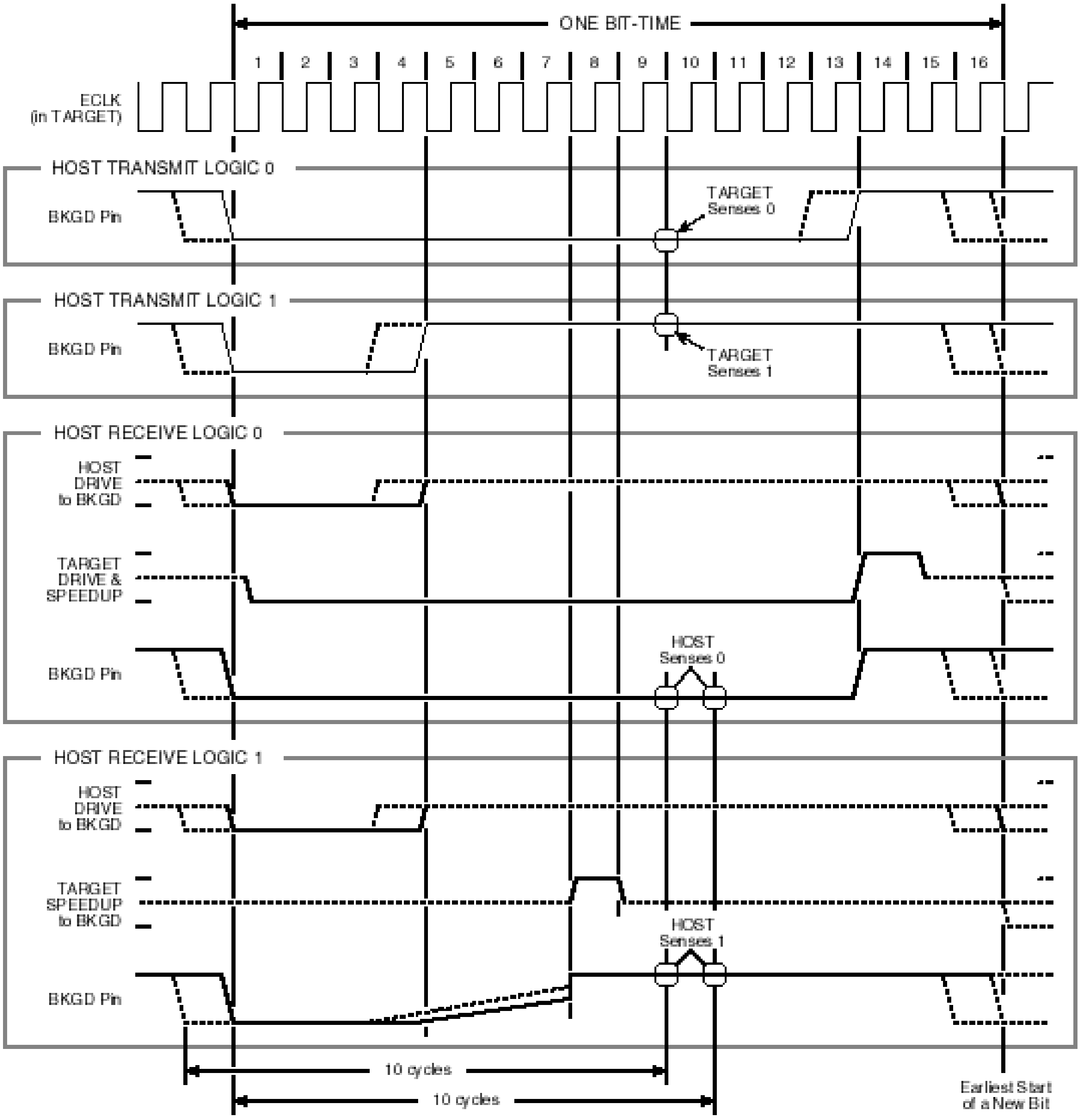


Figure 2 - Timing Details for Transmit and Receive Bit Times

When the host is sending data to the target, the speedup pulses are produced by the host (in fact the host could simply drive the pin high and low with an active driver). When the host is receiving data, the target is in control of the trailing (rising) edge on the BKGD pin so the target must generate the speedup pulses. The timing of this pulse is easy when the bit is a logic zero because the host has long since stopped driving the pin low. When the host is receiving a logic one, however, the target must wait long enough for the host to have stopped driving the pin actively low but must provide the driven-high speedup pulse before the host tries to sample the bit value. This speedup pulse is driven by the target at about cycle 8.

All of these types of cycles would occur during a READ_WORD command. First, the host would transmit the 8-bit command and a 16-bit address to the target MCU. A delay of up to 150 target E clocks is then imposed to be sure the BDM system has had time to find an unused bus cycle to perform the read. Then a 16-bit data word would be received from the target.

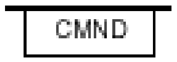
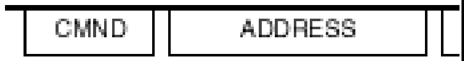
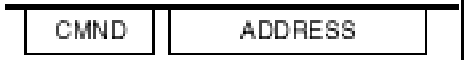
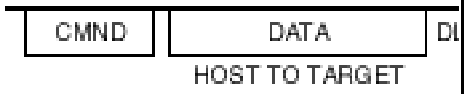
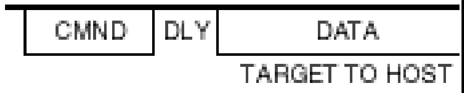
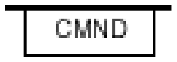
Two Types of Commands

In previous background systems, commands were processed by the main CPU which meant that debug operations and user application operations were mutually exclusive. This is the second type of commands in the M68HC12 and they are referred-to as firmware commands because they are controlled by a small firmware ROM. This ROM is only in the memory map while the background system is actually active so it has no effect on user applications. Using the main CPU to process these commands reduces the amount of logic needed to implement the background system. Commands in this group include read/write a selected CPU register, read/write the next word pointed-to by X and update X, trace one user instruction, go to user program, and tag-go.

The first type of commands called hardware commands, use a small block of dedicated logic and can be executed without interference to the user application. This type of command monitors the CPU bus activity and uses a cycle where the CPU does not require the busses. In cases where no such CPU free cycle can be found within a reasonable time, the background logic can momentarily stop the CPU to steal a cycle to perform the requested background function. It would take a very unusual sequence of instructions to keep the busses so busy that it is necessary to steal a cycle, and even then the impact on the application is much less than in conventional background debug systems. Commands in this group include commands to read or write a byte or word in user or BDM memory (total 8 memory access commands). The reason there are separate commands for user versus BDM memory is because the small BDM firmware ROM and a small number of BDM control registers have the same addresses as a portion of the user application memory. These BDM resources are only visible to the CPU while the background mode is actually active. There is also a hardware command to force the system into active background debug mode.

Table 1 summarizes the BDM commands and shows how they are structured. The simplest commands consist of an 8-bit command code sent from the host to the target.

Table 1 - Command Summary

Command Name	Code (hex)	Structure
Hardware Commands		
BACKGROUND	90	
WRITE_BYTE	C0	
WRITE_BD_BYTE	C4	
WRITE_WORD	C8	
WRITE_BD_WORD	CC	
READ_BYTE	E0	
READ_BD_BYTE	E4	
READ_WORD	E8	
READ_BD_WORD	EC	
Firmware Commands		
WRITE_NEXT	42	
WRITE_PC	43	
WRITE_D	44	
WRITE_X	45	
WRITE_Y	46	
WRITE_SP	47	
READ_NEXT	62	
READ_PC	63	
READ_D	64	
READ_X	65	
READ_Y	66	
READ_SP	67	
GO	08	
TRACE1	10	
TAG_GO	18	

Hardware commands that access memory in the target need to impose delays in case the target MCU has to wait for a free bus cycle to actually perform the access. When a hardware command attempts to access a target system memory location, the logic waits for a CPU free cycle where the CPU is not using the buses. Usually there would be a free cycle within a few cycles, but an unusual sequence of code with a lot of changes in flow could keep the buses busy. If the BDM logic does not find a free cycle within 128 cycles, the CPU is momentarily frozen to allow the BDM logic to steal a cycle to complete the access. If the access is a word access through a narrow 8-bit external bus, or if the access is to an area controlled by a stretched chip select space, the BDM can hold the CPU off for enough bus-rate cycles to complete the access.

The READ_NEXT and WRITE_NEXT (word) firmware commands offer the fastest way to read or write a block of target memory. Since the MCU is in active background mode when these commands are executed, there is no need to delay waiting for free cycles (although a shorter delay of about 32 E cycles is needed to allow BDM firmware to complete the requested access). Also these commands only require the 8-bit serial command plus the 16-bit read or write data. The speed of the BDM serial interface controls the best-case speed of data transfer (about 50 μs per word). These commands require the index register X in the target to be set to the first address in the block (-2) prior to starting the block transfer. As each word is accessed, X is pre-incremented by two.

The serial address and data transfers are always 16 bit pieces (commands are always 8 bits). For 8-bit sized accesses the host is responsible for writing or reading the correct half of the 16-bit data words. For example a byte read of address $FF01 will return the data in the low order half of the data word because the address was odd.

But Wait... It Also Programs Nonvolatile Memories

Although the background debug mode is primarily a debug tool, it also has important manufacturing and maintenance capabilities. Some versions of the M68HC12 have on-chip flash EEPROM and/or byte-erasable EEPROM. These memories can be programmed through the background system. No special provisions are needed in the end product except the ability to connect to ground and the single BKGD pin (and a high-voltage supply for flash programming/erase). Since the BKGD pin is also a mode select input during reset, such a system could be powered up in special single chip mode with the background system active for initial programming. Later when no programming system is connected to the BKGD pin, the target system can be powered up in normal single chip mode and execute the program that was loaded through the background system. You could even connect several target systems to a single programmer and program them in parallel.

If the application program ever needs to be changed, this same technique can be used to erase and reprogram the flash or EEPROM memories. This technique is not limited to on-chip memories. The background debug system can access anything that the application program can access including control registers, external memory systems, and any peripherals. This allows a variety of product test and calibration possibilities.

As an example of some of the capabilities of this system, consider a remote data sensing/logging system based on the M68HC12. The byte-erasable EEPROM can be used to store collected data samples. A data retrieval system could be connected to the data gathering system through a background connection. Previously gathered data could be read out of the gathering system without disturbing the ongoing data gathering function. The data retrieval system could then reinitialize software flags to let the gathering system know that the information has been collected and the memory is free to be used for new information.

In another application, target system performance could be monitored though the background connection by a portable maintenance system. If system adjustments are needed, the maintenance device could change software variables in the running application system. Unlike previous systems, all of the intelligence for this maintenance activity is in the portable device and no special interface software or hardware is needed in the application system. This is important because many application systems cannot afford any extra development effort, system memory, or the risk associated with incorporating such routines on top of their application system.

The Icing on the Cake

So far we have been talking about the basic background debug mode (BDM) found on all derivatives in the M68HC12 family. Some members of the family go a bit further by providing on-chip hardware breakpoint logic. This logic can be configured to provide a single match condition that includes 16-bits of address, 16-bits of data, and R/W. The comparators can also be configured to provide triggers on two separate addresses. In this dual address mode, the triggers can either produce tag signals that follow instructions through the instruction queue and stop the CPU just before the matched address is about to execute. Alternatively, this mode can be configured to generate a software interrupt (SWI) upon either address match.

In the dual address mode, the SWI option can be used to patch up to two software bugs in a ROM-based system. The breakpoint logic would be setup during initialization to produce an SWI when the program reached either of two error addresses. The SWI service routine could then call a patch routine in the EEPROM to replace the defective instructions. The last act of the repair code would be to jump back into the ROM to a point after the defective instructions.

The more obvious use for the hardware breakpoint feature would be for system debug. Software breakpoints can be used to debug code in a RAM by replacing selected instructions with an SWI or a background (BGND) instruction opcode. But this only works if the code is in a memory that can be changed by the debug system (i.e. a RAM or in some cases a byte-erasable EEPROM). A hardware breakpoint is more powerful because it can also be used to debug code in a ROM or flash EEPROM. To use the breakpoint logic for debug, you would configure it for dual address matching and set it to generate instruction tags. You could also use full address-data-R/W matching, to produce a background request. This mode would allow you to trigger on some condition other than an instruction. For example you may wish to stop if you ever write certain data to a certain address.

A Closing Remark

The HC12's full background system with hardware breakpoints provides a complete debug solution that rivals many full-feature hardware emulator systems - at a fraction of the cost. The more expensive hardware emulators can offer more than one or two breakpoints and more complex trigger logic. However some applications have very restricted physical access where the simple BDM connection is about all that can be connected.

In the next article in this series we will explore epanded memory systems for the HC12.

External Memory and the M68HC12 (part 3 of 3)

James M. Sibigtroth
Mail drop OE313
Motorola AMCU
6501 William Cannon Dr W
Austin, TX 78735-8598
RFTP70@email.sps.mot.com

The M68HC12 family is intended to cover a broad range of applications from single-chip systems with no external memory, to multi-megabyte 16-bit expanded systems. The first derivative, the MC68HC812A4, is intended for expanded systems and can operate in narrow (8-bit) non-multiplexed modes, or 16-bit non-multiplexed modes with up to 22 address lines. The second derivative, the MC68HC912B32, has 32 kbytes of flash, 1 kbyte of static RAM, and 768 bytes of byte-erasable EEPROM. The 'B32 is primarily intended to operate in single-chip modes, but it can be operated in a 16-bit multiplexed address/data mode (primarily used for factory testing).

An improved bank switching scheme is used to adapt the 16-bit address bus of the CPU to a 22-bit external address bus. New instructions in the CPU (CALL and RTC) simplify the programmer's job by eliminating the two most significant problems normally associated with bank switching. On-chip address translation logic and chip selects simplify the hardware designer's job.

Expansion Bus Basics

The HC12 has three primary modes, single-chip, expanded-narrow, and expanded-wide. There is also a special peripheral mode which is used for factory testing. All internal data busses are 16 bits but expanded-narrow mode allows a single external 8-bit program memory to be used to minimize system cost. In this mode, the CPU continues to see the data bus as 16 bits, but when a 16-bit word is accessed, the external bus interface momentarily freezes the CPU and performs two successive 8-bit accesses. In expanded-wide mode, the external data bus is 16 bits.

On-chip chip select circuits can even be configured to allow older 8-bit peripheral chips to be connected to one side of the data bus in an expanded-wide system. When the bus controller sees a word access to such a peripheral, it momentarily freezes the CPU while performing two 8-bit accesses.

Bus cycles on the HC12 work essentially the same as an HC11. The basic bus clock is the ECLK. The ECLK in an HC12 is typically half the frequency of the crystal. In an HC11 the ECLK was one fourth of the crystal rate. This change allows the HC12 to have lower oscillator current for a given bus frequency than the HC11. Bus cycles are said to begin at a falling edge on ECLK, and run to the next falling edge on ECLK. Addresses become valid during E-low and data is valid by the falling edge on ECLK. The R/W signal indicates data direction and has timing like an address line. A new control signal on the HC12 called LSTRB* (low strobe bar) indicates when the low half of the 16-bit data bus is valid. This is needed to support 8-bit accesses on a 16-bit bus. The decoding for R/W, address bit

0, and LSTRB* , is shown in Table 1.

Table 1. Decoding Bus Control Signals

R/W	A0	LSTRB*	Cycle Type
0	0	0	Write 16-bit Word
0	0	1	Write Even Byte
0	1	0	Write Odd Byte
0	1	1	Write Misaligned Word
1	0	0	Read 16-bit Word
1	0	1	Read Even Byte
1	1	0	Read Odd Byte
1	1	1	Read Misaligned Word

The misaligned word cases only occur when accessing the internal RAM. This RAM is specifically designed to perform misaligned word accesses within a single CPU bus cycle. Data appears swapped high byte for low byte, and an internal bus controller exchanges the high and low bytes before passing it on to the CPU12.

Expansion Beyond 64K

The HC12 architecture supports a bank select scheme that is tied into the CPU to overcome problems associated with traditional bank switching schemes. The CALL instruction saves the current program page number on the stack with the return address and then writes an instruction-supplied value to the bank select register and jumps to the called address. The return from call (RTC) instruction reverses the process to restore the old bank select value before returning. Since the bank select changes are done within uninterruptable instructions, there is no need to mask interrupts during the bank switch. Also, the code to change pages can be executed from within banked memory. This greatly reduces the programming effort to manage the bank select register.

Expansion of program space is done through a 16K window located from $8000 through $BFFF. The 8-bit PPAGE register allows up to 256 16K pages to be viewed through this window, one page at a time. Thus the HC12 architecture supports up to 4 megabytes of expanded program memory in addition to the other 48K of unpaged space outside the 16K program window. The PPAGE bank select register is accessible as an ordinary control register but the CPU12 can access this register using hardware control lines so the CPU does not need too know the address of this register. This allows new derivatives to be developed with registers in a different location without having to make any changes to the CPU12.

Two other expansion windows are provided for data expansion. A 4K window form $7000 through $7FFF allows up to one megabyte of paged data memory in 256 4K pages selected by the DPAGE register. A 1K window allows an "extra" data expansion area that may be located at $0400-$07FF or at $0000-$03FF which includes direct addressing space. The EPAGE register selects one of 256 1K pages to be viewed through this window, one at a time. Although you could put executable programs in these spaces, it would take more programming effort because the management of the DPAGE and EPAGE registers would need to be done using traditional bank select algorithms. These algorithms require more care and programming overhead than the CALL/RTC mechanism associated with PPAGE.

On-Chip Address Translation Logic

The CPU12 itself, always generates 16-bit addresses, but the external address bus can be up to 22 bits. Internal address translation logic takes care of multiplexing bank select values onto the higher order address lines at appropriate times. When an address does not fall within any enabled expansion window, ones are driven on the external address lines A21-A16. When an address is in the $8000 - $BFFF range and the program expansion window is enabled, the address translation logic multiplexes the PPAGE value onto external addresses A21-A14. When an address falls within the $7xxx range and the data expansion window is enabled, the address translation logic multiplexes the DPAGE value onto external addresses A19-A12 (A21 and A20 are driven to ones). When an address falls within the extra expansion window (and it is enabled), the address translation logic multiplexes the EPAGE value onto external addresses A17-A10 (A21 - A18 are driven to ones).

This address translation scheme does not always provide the external system with complete information about the address. When an expansion page register (PPAGE, DPAGE, or EPAGE) is multiplexed onto the external address lines, some of the internal CPU address lines become hidden from the external system. For example, when PPAGE is multiplexed onto A21 - A14, CPU address lines A15 and A14 are not visible to the external system. A CPU access to $FFFF in unpaged space, and an access to $BFFF on PPAGE $FF, both produce the same 22-bit address, $3FFFFF. In the case of chip select CSP0, this is beneficial because it allows a single external program memory to include the unpaged space from $C000 - $FFFF (which includes the interrupt vectors) and several pages of expansion memory in the paged space from $8000 - $BFFF. The programmer must simply remember not to try to use the same physical memory for both unpaged space, and the highest PPAGE block.

In the case of DPAGE or EPAGE, problems can be avoided by using chip selects CSD and CS3 respectively. These chip selects have access to the internal address lines so they are only active for addresses within the desired expansion window. Even though the 22-bit external addresses could be the same for two different internally generated accesses, the chip selects would differentiate them.

In some systems, high order address lines can be used to differentiate unpaged accesses and paged accesses. In unpaged accesses, external address lines A21 - A16 are always ones. For paged accesses to the DPAGE or EPAGE areas, the corresponding page register is multiplexed onto some of these lines. Suppose you have a 32 kbyte static RAM connected with its address lines MA14 - MA0 connected to external address lines A14 - A0.

Figure 1 shows 22-bit external MCU addresses along the left side of the figure. The address space from $000000 - $2FFFFF corresponds to pages $00 - $BF in the PPAGE expansion area. No access to internal memory space, DPAGE, or EPAGE can generate these external addresses. DPAGE $00 is at MCU addresses $300xxx. Expansion data pages (DPAGEs) are located in the MCU memory map from $300000 - $3FFFFF. These DPAGEs are blocked off in groups of eight 4K pages in this figure, because this discussion assumes an external 32 kbyte RAM. Before considering chip enable inputs to the RAM, this RAM would appear in the memory map of the MCU in several places. This is called mirroring and it occurs whenever some address lines are not considered in an address decode.

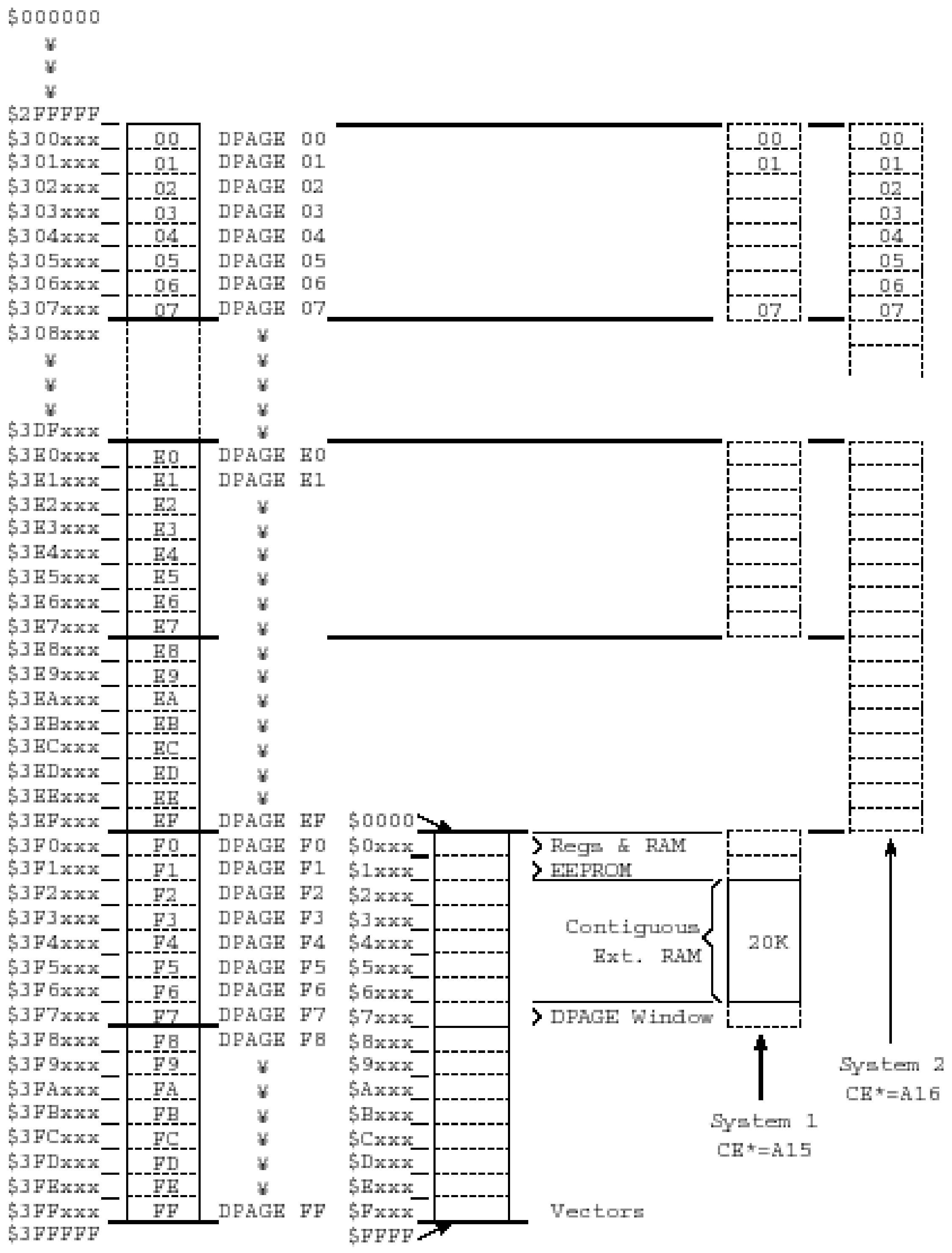


Figure 1. Memory Mapping Example

In the lower middle portion of the figure the 64K CPU address space is shown. This space corresponds to MCU addresses $3F0000 - $3FFFFF because the upper six external address lines are forced to ones for unpaged accesses. Internal MCU register space and internal RAM are located within the first 4K of CPU address space ($0xxx). On-chip EEPROM fills the next 4K ($1xxx). The DPAGE window fills $7xxx and vectors are located in the $FFxx space. Internal resources take priority over all external or paged memory accesses. The right portion of the figure shows two alternate ways of configuring the external 32K RAM for application use.

If external address line A15 is connected to the low-true chip enable for the external RAM, it appears in every other 32K block starting at $300000. The user can choose to access the 32 kbytes of physical memory as DPAGEs 00, 01, and 07, and as a separate contiguous 20K block from $2000 - $6FFF in unpaged space. This could be more useful than using the external RAM as eight 4K pages.

If external address line A16 is connected to the low-true chip enable for the RAM, the RAM appears in every other 64K block starting at $300000. This causes the external RAM to be disabled for all 64K of unpaged CPU address space. In an application the programmer would now use the external RAM as DPAGEs 00 through 07. Other DPAGE values such as E0 could be used, but they would redundantly access some portion of the 32 kbytes of the physical RAM. For example DPAGE E0 is the same physical memory as DPAGE 00 because of the way we chose to connect the RAM.

In our hypothetical system, external address lines A21 - A17 were not used. Control bits in the HC12 MCU allow you to configure the pins associated with any combination of address lines A21 - A16 as general purpose I/O pins when they are not needed as expansion address lines.

Closing Remarks

The three articles in this series have touched on some of the most interesting features of Motorola's new M68HC12 family. In the first article we discussed the new 16-bit CPU. The CPU12 helps a new generation of C compilers produce fast efficient code while keeping compatibility with the M68HC11 family. The second article explained the single-wire background debug system. This allows easy access to the inner workings of a target application system without disturbing the application program or using its resources. In this third article we looked at the flexible expansion bus structure. This allows a user to choose single chip operation for the maximum number of I/O pins, a cost-effective narrow bus to allow a single 8-bit program memory, or a full 16-bit external data bus for maximum performance. For those who need more than 64 kbytes of memory, the M68HC12 offers a code efficient solution that is easier to use than traditional bank switching systems. To learn more about the M68HC12, try Motorola's web site at http://www.mcu.motsps.com or contact a Motorola representative.